
Contents

About the Editor, xv

Acknowledgments, xvi

Zeba Academy – Mastering Computer Science, xvii

CHAPTER 1 ■ Introduction	1
EVOLUTION OF PERL	1
WHY PERL?	2
PERL FEATURES	3
PERL APPLICATION	4
PERL IMPLEMENTATION	4
PROGRAMMING IN PERL	4
Comments	5
Perl's Benefits	6
Perl's Disadvantages	6
Applications	6
PERL INSTALLATION AND CONFIGURATION ON WINDOWS, LINUX, AND MACOS	7
PERL DOWNLOAD AND INSTALLATION	8
HELLO WORLD PROGRAM	9
HOW TO EXECUTE A PERL PROGRAM	11
Windows	11
Unix/Linux	11
A PERL PROGRAM'S BASIC SYNTAX	11
Variables	12

Expressions	12
Comments	13
Statements	14
Block	14
Functions or Subroutines	15
Loops	15
Whitespaces and Indentation	16
Keywords	17
DATA TYPES	18
Scalars	18
Arrays	18
Hashes	19
NOTE	20
CHAPTER 2 ■ Fundamentals of Perl	21
PERL CODE WRITING METHODS	21
Interactive Mode	22
Script Mode	23
One-Liner Mode	24
BOOLEAN VALUES IN PERL	25
OPERATORS	29
Arithmetic Operators	30
<i>Addition</i>	30
<i>Subtraction</i>	30
<i>Multiplication</i>	30
<i>Division</i>	30
<i>Modulus</i>	31
<i>Exponent Operator</i>	31
Relational Operators	31
Logical Operators	34
Bitwise Operators	35

Assignment Operators	37
Ternary Operator	40
VARIABLES IN PERL	41
Naming of a Variable	41
Declaration of a Variable	42
Modification of a Variable	42
Variable Interpolation	43
VARIABLES AND ITS TYPES	44
Creating Variables	45
Scalar Variables	45
Array Variables	46
Hash Variables	46
Variable Context	47
SCOPE OF VARIABLES	48
The Scope of Global Variables	48
Lexical Variables' Scope (Private Variables)	50
Package Variables	52
MODULES IN PERL	54
Making a Perl Module	54
Importing and Using a Perl Module	55
Utilizing Module Variables	55
Making Use of predefined Modules	56
PERL PACKAGES	57
Perl Module Declaration	57
Making Use of a Perl Module	58
Using a Different Directory to Access a Package	58
Utilizing Module Variables	59
Begin and End Block	60
NUMBER AND ITS TYPES IN PERL	60
DIRECTORIES WITH CRUD OPERATIONS IN PERL	64
Making a New Directory	65
Opening an Existing Directory	65

Read Directory in the Scalar and List Context	65
Modifying Directory Path	67
Directory Closing	68
Delete a Directory	68
NOTES	69
CHAPTER 3 ■ Input and Output in Perl	71
PERL print() AND say() METHODS	71
print() Operator	71
say() Function	73
print OPERATOR	73
USE OF STDIN FOR INPUT	74
CHAPTER 4 ■ Control Flow in Perl	77
DECISION-MAKING IN PERL	77
if Statement	78
if else Statement	79
Nested if Statement	81
if elsif else ladder Statement	83
unless Statement	85
unless else Statement	86
unless elsif Statement	88
LOOPS IN PERL	90
for Loop	90
foreach Loop	92
while Loop	92
Infinite While Loop	93
do...while loop	93
until Loop	94
Nested Loops	95
given-when STATEMENT	97
Nested given-when Statement	98
goto STATEMENT	100

next OPERATOR	103
redo OPERATOR	105
last IN LOOP	106
NOTES	107
 CHAPTER 5 ■ File Handling in Perl	 109
INTRODUCTION OF FILE HANDLING	109
Using FileHandle To Read and Write to a File	110
Various File Handling Modes	111
Redirecting Output	115
FILE OPENING AND READING	116
Opening a File	117
Reading a File	117
<i>FileHandle Operator</i>	117
<i>getc Function</i>	118
<i>read Function</i>	118
Reading More than One Line at a Time	119
Exception Handling in Files	119
<i>Throw an Exception</i>	119
<i>Give a Warning</i>	120
WRITING TO A FILE	120
print() Function	120
Error Handling and Error Reporting	122
<i>Throw an Exception (Using Die Function)</i>	122
<i>Give a Warning (Using Warn Function)</i>	123
APPENDING TO A FILE	123
CSV FILE READING	125
Use of Split() for Data Extraction	125
Character Escaping a Comma	127
Installation of the TEXT::CSV	128
Fields with Newlines Embedded	129
FILE TEST OPERATORS	131

FILE LOCKING	133
flock()	134
flock() vs lockf()	136
SLURP MODULE	136
USEFUL FILE-HANDLING FUNCTIONS	139
CHAPTER 6 ■ Regular Expressions in Perl	141
OPERATORS IN REGULAR EXPRESSION	143
REGEX CHARACTER CLASSES	147
SPECIAL CHARACTER CLASSES IN REGULAR EXPRESSIONS	150
QUANTIFIERS IN REGULAR EXPRESSION	152
Quantifier Table	153
BACKTRACKING IN REGULAR EXPRESSION	156
BACKTRACKING	158
“e” MODIFIER IN REGULAR EXPRESSION	159
The Substitution Operation Is Performed using a Subroutine	160
REGEX “ee” MODIFIER	161
When Doing Mathematical Calculations, Use the “ee” Modifier	162
pos() FUNCTION IN REGULAR EXPRESSION	164
To Match from a Specified Position, use \G Assertion	166
REGEX CHEAT SHEET	167
Character Classes	168
Anchors	169
Metacharacters	170
Quantifiers	170
Modifiers	171
White Space Modifiers	171
Quantifiers – Modifiers	172
Grouping and Capturing	172
SEARCHING IN A FILE USING REGEX	172
Regular Search	173

Using Word Boundary in the Regex Search	174
Use of Wildcards in the Regular Expression	175
CHAPTER 7 ■ Object-Oriented Programming in Perl	177
CLASSES IN OOP	181
Object	181
Class	181
Data Member	182
Defining a Class	182
Creating a Class and Making Use of Objects	182
Creating a Class Instance	182
Creating an Object	183
OBJECTS IN OOPs	184
METHODS IN OOPs	186
Types of Methods in Perl	187
get-set Methods	187
CONSTRUCTORS AND DESTRUCTORS	189
Constructors	189
Passing Dynamic Attributes	191
Destructors	192
METHOD OVERRIDING IN OOPs	192
Why Do We Override Methods?	196
INHERITANCE IN OOPs	196
Base Class and Derived Class	197
Multilevel Inheritance	198
Implementing Inheritance in the Perl	199
POLYMORPHISM IN OOPs	200
ENCAPSULATION IN OOPs	203
NOTE	205
CHAPTER 8 ■ Subroutines in Perl	207
SUBROUTINES OR FUNCTIONS	207
Determining Subroutines	208

Calling Subroutines	208
Passing Parameters to Subroutines	208
Passing Hashes to Subroutines	210
Passing Lists to Subroutines	210
Returning a Value from a Subroutine	211
Local and Global Variables in Subroutines	212
A Varying Number of Parameters in a Subroutine Call	213
FUNCTION SIGNATURE IN PERL	214
Defining Subroutines	214
Function Signature	214
Passing Parameters of a Type other than that Specified in the Signature	215
Difference in Number of Arguments	216
PASSING COMPLEX PARAMETERS TO A SUBROUTINE	217
MUTABLE AND IMMUTABLE PARAMETERS	222
Mutable Parameters	222
Immutable Parameters	222
Traits	223
MULTIPLE SUBROUTINES	225
Subroutine Definition	225
Use of the “multi” Keyword	226
return() FUNCTION	228
REFERENCES IN PERL	229
Making a Reference	229
Dereferencing	231
PASS BY REFERENCE	232
PERL RECURSION	234
APPRAISAL, 237	
BIBLIOGRAPHY, 281	
INDEX, 287	

Introduction

IN THIS CHAPTER

- Perl Introduction
- Installation and Environment Setup
- Hello World Program

Perl is a high-level, interpreted, and dynamic, general-purpose programming language.

It was created in 1987 by Larry Wall. Although “Practical Extraction and Reporting Language” is the most common expansion, there is no official full form of Perl. Some programmers refer to Perl as “Pathologically Eclectic Rubbish Lister” and “Practically Everything Likable.” The abbreviation “Practical Extraction and Reporting Language” is often used since Perl was initially designed for text processing, such as extracting the necessary information from a specific text file and converting the text file to a new format. Perl supports both procedural and object-oriented programming.

Perl is syntactically similar to C, making it accessible to C and C++ programmers.

EVOLUTION OF PERL

It all began when Larry Wall was tasked with generating reports from several text files containing cross-references. Then he began using `awk` for this work but quickly realized it was insufficient. Instead of designing a tool for this purpose, he created a new programming language, Perl, and its

interpreter. He built the Perl programming language in C, using principles from awk, sed, and LISP, among others. Perl was initially designed only for system administration and text processing, but subsequent versions included the ability to manage regular expressions (Regex) and network sockets. Presently, Perl is renowned for its capability to handle Regex. The first release of Perl was version 1.0 on December 18, 1987. 5.28 is the most recent version of Perl. Perl 6 is distinct from Perl 5 since it is a reimplementation of Perl 5 that is entirely object-oriented.

WHY PERL?

Perl's popularity and demand stem from a variety of factors. Some of the causes are as follows:

- Perl is a high-level language; thus, it is similar to other popular programming languages such as C and C++, making it simple for anybody to learn.
- Text-processing: As the name “Practical Extraction and Reporting Language” suggests, Perl has powerful text manipulation capabilities, allowing it to create reports from various text files readily. Additionally, it may convert the files to a different format.
- Perl incorporates the most exemplary aspects of other languages, such as C, sed, awk, and sh, making the language more useful and productive.
- System management: Perl's ability to use many scripting languages makes system administration a simple operation. Instead of relying on many languages, utilize Perl to fulfill all system management tasks. Despite this, Perl is also used for web development, web automation, and Graphical User Interface (GUI) programming, among other things.
- Web and Perl: Perl may be incorporated into web servers to boost their processing capacity, and the DBI module makes web-database interaction quite simple.

Getting started with Perl programming:

- Locating an interpreter: Numerous online integrated development environments (IDEs) may be used to execute Perl programs without installation.

- Windows: A number of IDEs are available to execute Perl applications or scripts, including Padre and Eclipse with the EPIC plugin.

PERL FEATURES

- The capabilities listed below are present in Perl and have been widely adopted by other programming and scripting languages.
- Most of Perl's features, including variables, expressions, statements, control structures, and subroutines, are derived from C.
- It also utilizes shell scripting tools for detecting data kinds.
- Perl provides built-in functions often used in shell programming, such as sort and system facilities use, that may be denoted using leading sigils, such as an array, scalar, and hash.
- In addition, Perl 5 supports complicated data structures and an object-oriented programming style that includes packages, references, and compiler directives.
- The interpreter knows the storage and memory needs for each data type and allocates and deallocates memory depending on use in all versions of Perl.
- It also does typecast during execution, such as converting an integer to a string, and other conversions that are not valid, which result in errors being raised.
- Perl does not impose or promote any particular programming style, such as procedural, object-oriented, or functional; the interpreter and its functions constitute the language's only definition.
- Perl has APIs (text manipulation facilities) that help deal with XML, HTML, and other mark-up languages.
- Perl offers the most significant level of security and is even certified by Coverity, a third-party security company, for having a low defect density and fewer security issues.
- Perl is extensible and includes libraries that handle XML and database integration (DI), such as Oracle and MySQL.

PERL APPLICATION

Perl, along with Hypertext Preprocessor (PHP) and Python, is a popular programming language among developers. Previously, programmers wrote Common Gateway Interface (CGI) scripts using Perl. Perl is frequently used as a departmental adhesive between heterogeneous and non-seamless interoperable systems. System administrators adore this language because they can enter a single command to accomplish a task that would otherwise require writing a program. Perl is primarily portable, with some Windows and macOS-specific customizations.

Additionally, developers use the language to build and deploy. It is used by most suppliers or software manufacturers to package and deploy commercial software (including COTS and bespoke). It is utilized extensively in the fields of finance and bioinformatics due to its capacity to manage and process large data sets.

PERL IMPLEMENTATION

As stated before, Perl is an interpreted language written in C with an extensive library of modules written in both C and Perl. The Perl interpreter comprises a massive 150,000 lines of C code, which compiles to 1 MB on most system architectures. There are almost 500 modules in the Perl distribution, including 300,000 lines of Perl code and 200,000 lines of C code. The components of Perl (arrays, scalars, and hashes) are represented as C structures inside the object-oriented design of the interpreter.

The interpreter's life cycle consists of two phases: compilation and execution. The interpreter parses the Perl code into a syntax tree at build time. It executes the Perl program by traversing the tree at runtime. The Perl programming language is offered as open source with 120,000 functional tests; the interpreter and other functional modules are rigorously tested during the compilation process. If these 120K functional tests pass, it is safe to assume that your code will not damage the interpreter.

PROGRAMMING IN PERL

Perl is syntactically similar to other commonly known programming languages, making it easy to write and understand. Perl programs can be written in any of the most popular text editors, including Notepad++ and gedit. After creating the program, save the file with the .pl or .PL extension. Type Perl file name.pl on the command line to execute the program.

Example: A simple application to print Welcome to PFP!

```
# Program to print Welcome to PFP!
#!/usr/bin/perl

# Below line will print "Welcome to PFP!"
print "Welcome to GFG!\n";
```

Comments

Comments are used to improve code readability. The comment items are ignored by the interpreter and are not executed. Comments might be single or multiple lines.

- Single line Comment

Syntax:

```
# Single-line-comment
```

- Multi-line comment

Syntax:

```
= Multi-line comments
Line start from = is interpreted as
starting of multiline comment and =cut is
consider as the end of the multiline comment
=cut
```

In the above example:

1. **print:** It is a Perl function that displays the result or any provided output on the console.
2. **Quotes:** In Perl, we can use either single or double quotes (" or "). Single quotes do not interpolate any variable or special character, but double quotes do.
3. **\n:** It is used for newline character, which escapes any character with a backslash (\).
4. **/usr/bin/Perl:** It is the original Perl interpreter binary, which always begins with #!. This is used in Perl Script Mode Programming.

5. Because Perl is a case-sensitive computer language, `$Geeks` and `$geeks` are distinct identifiers.

Perl's Benefits

- Perl supports cross-platform compatibility and mark-up languages such as HTML, XML, and others.
- It is particularly efficient in text manipulation, i.e. Regex. It also has socket capabilities.
- Free and open-source software is released under the Artistic and GNU General Public Licenses (GPL).
- Because it is an embeddable language, it can be used in web servers and database servers.
- It supports around 25,000 open-source modules on CPAN (Comprehensive Perl Archive Network), which provide numerous useful enhancements to the standard library. For example, XML processing, GUI, and DI, among others.

Perl's Disadvantages

- Because of CPAN modules, Perl does not support portability.
- Programs run slowly and must be interpreted each time changes are made.
- In Perl, the same result may be obtained in a variety of methods, making the code messy and illegible.
- When compared to other languages, the usability factor is lower.

Applications

- Text file processing and string analysis are key uses of the Perl language.
- Perl is also used for CGI scripts.
- Used in web development and GUI design.
- The text-handling features of Perl are also employed to generate SQL queries.

PERL INSTALLATION AND CONFIGURATION ON WINDOWS, LINUX, AND MACOS

Installing Perl on our system, whether it is Windows, Linux, or Macintosh, is the first step. We must have a direct understanding of what the Perl programming language is and what it accomplishes.

Perl is a dynamic, high-level, interpreted, general-purpose programming language. Perl was first designed for text processing, such as extracting the necessary information from a particular text file and converting text file to a new format. Perl supports both procedural and Object-Oriented programming. Perl is syntactically similar to C, making it accessible to C and C++ programmers.

Programming in Perl is possible using any plain text editor, such as notepad, notepad++, or any other application. One may either utilize an online IDE or install one on their machine to make creating Perl routines easier. Using an IDE makes it simpler to create Perl code due to the IDE's many capabilities, such as an intuitive code editor, debugger and compiler. Perl must be installed on a system before developing Perl codes and executing many exciting and beneficial operations. This may accomplish by following the detailed methods given below:

Checking for a preinstalled version of Perl: Before we begin the installation of Perl, it is a good idea to check if it is already installed on your system. Many software applications require Perl to perform their operations, so a version of Perl may include in the software's installation package. There is no need to redownload and reinstall Perl if this is the case.

Macintosh also preinstalls Perl on its computers. Perl is preloaded on many Linux systems. Simply use the command prompt to see whether Perl is preloaded on your device.

(For Windows, type cmd in the Run dialogue (window + R), for Linux, press Ctrl+Alt+T, and for macOS, press Control+Option+Shift.)

Execute the following command now:

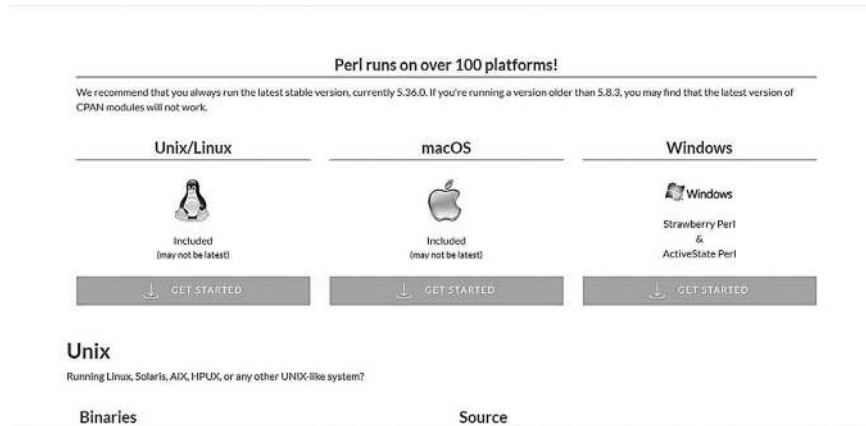
```
perl -v
```

If Perl is already installed, a message with all the details of Perl's version will be generated; otherwise, an error stating bad command or file name will be generated.

PERL DOWNLOAD AND INSTALLATION

Perl download:

- Before starting with installation process, we need to download it. For that, all versions of Perl for the Windows, Linux, and macOS are available on <https://www.perl.org/get.html>



Perl download.

- Install Perl by downloading it and following the installation instructions.

Starting with the Windows installation:

- Completing the User's License Agreement.
- Deciding what to install.
- The installation procedure.
- Completed installation.

After finishing the installation procedure, any IDE or text editor may use to develop Perl codes, which can then be run on the IDE or the command prompt with the command:

```
perl file1.pl
```

Starting with the Linux installation:

- Change the path to install Perl.
- Start the installation process.
- Choosing the Perl Installation Directory.
- Finishing the installation.
- Once the installation procedure is complete, any IDE or text editor may be used to create Perl codes, which can then launch on the IDE or the command prompt with the command:

```
perl file1.pl
```

Starting with the macOS installation:

- Starting out.
- Finished with the User's License Agreement.
- Choosing a place for installation.
- The installation procedure.
- Installation is now complete.
- After finishing the installation procedure, any IDE or text editor may use to develop Perl codes, which can then be run on the IDE or the command prompt with the command:

```
Perl file.pl
```

HELLO WORLD PROGRAM

Perl is a computer language explicitly developed for text processing. The acronym stands for Practical Extraction and Report Language. It is compatible with several systems, including Windows, Mac OS, and almost all UNIX variants. The Hello Everyone! program in each programming language provides novice programmers with a head start in learning the language.¹

The result of the simple Hello everyone program is “Hello Everyone!” printed on the screen.

A simple Perl program consists of the following execution steps:

- **Step 1:** Transfer the file to the Perl interpreter.

In Perl, the first line always begins with the pair of characters `#!`. It instructs the Perl interpreter on how to execute the file. The file should relocate to the `/usr/bin/Perl` subdirectory, which contains the Perl interpreter.

Therefore, the first line of the program will read as follows:

```
#!/usr/bin/perl
```

- **Step 2: Perl Pragma**

A pragma is a special module in the Perl package that has control over various features of Perl's compile time or run time behavior, such as strictness or warnings. So the following two lines are as follows:

```
use strict;
use warnings;
```

- **Step 3: Make use of the `print()` function.**

Finally, we use Perl's `print()` method to display a string to display the output.

```
print("Hello Everyone\n");
```

Example:

```
#!/usr/bin/perl

# Modules-used
use strict;
use warnings;

# Print function
print("Hello Everyone\n");
```

Important points:

- The file must save with the extension `.pl`.
- The Perl package directory must be the same as where the program file is saved.

HOW TO EXECUTE A PERL PROGRAM

In general, there are two methods for running a Perl program: using online IDEs: We can run Perl programs without installing them by using various online IDEs.

Using the command line: We can launch a Perl program using command line options. The following instructions explain how to launch a Perl program from the command line in Windows/Unix:

Windows

To begin, launch a text editor such as Notepad or Notepad++.

Write the code in a text editor and save it as a .pl file.

Ascertain that we have obtained and installed the most recent Perl version from <https://www.perl.org/get.html>.

Open the command line and type `Perl -v` to see if the newest version of Perl has been correctly installed.

Type `Perl HelloEveryone.pl` to build the code. If our code has no errors, it will run correctly and the output will be shown.

Unix/Linux

Follow the steps above to write code and save the file with the .pl extension.

To download and install Perl, use the terminal application on our Unix/Linux operating system and follow the steps below.

Now, execute the command `Perl -version` to see if the newest version of Perl was correctly installed.

Type `Perl hello.pl` to build the code. If our code has no errors, it will run correctly, and the output will show.

A PERL PROGRAM'S BASIC SYNTAX

Perl is a dynamic, high-level, interpreted, general-purpose programming language. Perl was first designed for text processing, such as extracting the necessary information from a particular text file and converting text file to a new format. Perl supports both procedural and object-oriented programming.

Perl is similar to C, making it accessible to C and C++ programmers. Perl follows a basic syntax for developing programs for applications and software and writing simple Perl programs, similar to those of other programming languages.

This grammar features several predefined phrases such as keywords, variables for storing values, expressions, statements for performing logic,

loops for iterating through a variable value, blocks for combining statements, and subroutines for simplifying code.

All of these elements, when combined, constitute a Perl program. The variables, statements, and other factors that make up a program's syntax are used by every Perl program, whether it's a simple code for adding two numbers or a complex one for launching web scripts.

Variables

Variables are user-defined words used to store the program's values and utilized to evaluate the code. Every Perl program comprises values on which the code operates. These values cannot be edited or saved unless a variable is used. A value can only process if it is saved in a variable and the variable's name is used.

A value is data that is passed to the program to conduct a modification action. This data can be numbers, strings, characters, lists, etc.

Example:

Values:

15

peeks

25

Variables:

`$c = 15;`

`$d = "peeks";`

`$e = 25;`

Expressions

Variables and an operator symbol make up Perl expressions. These expressions define the operation performed on the data given by the relevant code. In Perl, an expression is something that, when evaluated, returns a value. An expression can also be a value without any variables or operator symbols. It can be either an integer or a string with no variables.

Example:

Value 10 is an expression, `$c + $d` is an expression that returns their sum, etc.

A more complex expression is one that uses Regex to perform operations on strings and substrings.

Comments

Perl developers frequently utilize the comment system since things may quickly become complicated without it. Comments are essential information provided by developers to help the reader comprehend the source code. It describes the reasoning or a portion of it that is employed in the code. When we are no longer available to address queries about our code, comments are frequently helpful to someone who is maintaining or improving it. These are frequently noted as a good programming practice that does not affect program output but increases overall readability.

In Perl, there are two kinds of comments:

1. **Single-line comments:** A single-line comment in Perl begins with the hashtag symbol (#) and continues until the end of the line. If the comment is more than one line, add a hashtag to the following line and continue the comment. Single-line comments in Perl have proven helpful in providing brief explanations for variables, function declarations, and expressions. The following code for an example of a one-line comment:

```
#!/usr/bin/perl
$d = 10; # Assigning value to $d
$e = 30; # Assigning value to $e

$c = $d + $e; # Performing operation
print "$c"; # Printing result
```

2. **A multi-line string as a comment:** A multi-line comment in Perl is a chunk of text enclosed by “=” and “=cut.” They are useful when the remark content does not fit on a single line and must spread across lines. Multi-line comments or paragraphs provide documentation for people who are reading your code. Perl considers everything typed after the “=” sign to be a comment until a “= cut follows it.” Please remember that there should be no whitespace following the “=” symbol. See the following code snippet for an example of a multi-line comment:

```
#!/usr/bin/perl

=Assigning values to
The variable $d and $e
=cut
```

```

$d = 10;
$e = 30;

=Performing operation
and printing result
=cut
$c = $d + $e;
print "$c";

```

Statements

A statement in Perl contains instructions for the compiler to conduct operations. These statements conduct the operations on the variables and values at the run-time. Every sentence in Perl must conclude with a semicolon(;). Basically, instructions put in the source code for execution are termed statements. There are several sorts of statements in the Perl programming language, such as assignment statements, conditional statements, and looping statements. These all allow the user to acquire the appropriate output. For example, `n = 60` is an assignment statement.

Multi-line statements: Statements in Perl may be expanded to one or more lines by simply separating them into pieces. Unlike other languages like Python, Perl looks for a semicolon to terminate the sentence. Every line between two semicolons is treated as a single sentence.

When the programmer wants to execute extensive computations and cannot fit his statements onto one line, one may simply split it into numerous lines.

Example:

```

$z = $g + $h + $i +
    $j + $k + $l;

```

Block

A block is a collection of statements that execute a related operation. Multiple statements in Perl can be performed concurrently (under a single condition or loop) by using curly braces ({}). This creates a block of statements that are all performed simultaneously. This block can use to optimize the program by grouping statements together.

Variables declared within a block have a scope confined to that block and are useless outside of it. They will only be executed when that specific block is being run.

Example:

```
{
    $d = 25;
    $d = $d + 35;
    print($d);
}
```

The variable `$d` in the preceding code has a scope confined to this specific block and is useless outside of it. The block above contains statements with operations that are related to one another.

Functions or Subroutines

A function/subroutine is a code block written in a program to execute a specified task. To better understand how functions work, we may compare functions in programs to people in a real-world workplace. Assume the manager assigns his staff the task of calculating the annual budget. So, how will this procedure be completed? The employee will obtain statistics information from the employer, run calculations, compute the budget, and provide the results to his supervisor. Functions operate similarly. They accept information as an argument, run a set of statements, conduct operations on these parameters, and then return the output.

Perl offers two primary types of functions:

- Built-in functions: Perl has a large number of built-in library functions. These functions have already been coded and saved as functions. To use them, we just need to call them when needed, such as `sin()`, `cos()`, `chr()`, `return()`, and `shift()`.
- User-defined functions: Perl allows us to write our own customized functions, known as user-defined functions or subroutines, in addition to the built-in functions.

Using this, we may develop our code packages and call them whenever we need them.

Loops

Like any other language, Perl uses loops to execute a statement or a group of instructions repeatedly until and unless a specific condition is fulfilled. This allows the user to save time and effort by not writing the same code multiple times.

Perl offers several looping techniques:

- for loop
- for each loop
- until loop
- while loop
- do...while loop
- nested loops

Whitespaces and Indentation

In Perl, whitespaces are blank spaces used between variables and operators, between keywords, and so on. Whitespaces have no effect in Perl unless quote marks surround them. Whitespaces such as spaces, tabs, and newlines have the same meaning in Perl when used outside quotes.

First example:

```
$c = $d + $e;
Here, spaces are of no use,
it will cause no effect even if it is written as
$c = $d + $e;
```

Second example:

```
print "Peeks for Peeks";
will print
Peeks for peeks
whereas,
print "Peeks      for
                peeks";
will print
Peeks      for
                Peeks
```

In the preceding instances, whitespaces have no impact unless placed within quotations.

Similarly, indentation is used to structure the code to make it easier for users to read. When a block of statements is utilized, indentation helps to reduce the code's readability difficulty.

Example:

Using Indentation:

```
{
    $c = $d + $e;
    print "$a";
}
```

Without using Indentation:

```
{
$c = $d + $e;
print "$c";
}
```

In the above example, both blocks will function identically; but indentation makes it more reader-friendly for scripts with a significant number of statements.

Though using whitespaces and indentation in your Perl code is not required, it is a recommended practice.

Keywords

Keywords, often known as reserved words, are terms in a language used for internal processes or to indicate predefined actions. They have a unique significance for the compiler. As a result, these terms are not permitted to be used as variable names or objects. This will produce a compile-time error. Along with control words, keywords in Perl contain built-in functions.

These keywords can occasionally be used as variable names, although this causes confusion and makes debugging such a program difficult.

Example:

One can use `$print` as a variable and keyword `print()`.

DATA TYPES

Data types define the sorts of data that can store in a valid Perl variable. Perl is a loosely typed programming language. There is no need to provide a data type while using the Perl program. The Perl interpreter will choose the type based on the context of the data.

Perl has three data types, which are as follows:

- Scalars
- Arrays
- Hashes

Scalars

A single data unit can be an integer, a floating-point value, a character, a string, a paragraph, or an entire web page.

Example:

```
# Program to demonstrate the
# Scalars data types

# integer assignment
$age = 1;

# string
$name = "ABC";

# floating point
$salary = 21.5;

# displaying result
print "Age is = $age\n";
print "Name is = $name\n";
print "Salary is = $salary\n";
```

Arrays

Array is a variable that stores values of the same data type as a list. In Perl, we use the “@” symbol before the variable name to declare an array.

```
@age=(20, 30, 40)
```

It will generate an array of integers with the values 20, 30, and 40. The “\$” symbol is used to access a single member in an array.

```
$age[0]
```

Example:

```
# Program to demonstrate
# the Arrays data type

#!/usr/bin/perl

# creation of the arrays
@ages = (43, 21, 29);
@names = ("Peeks", "for", "Peeks");

# displaying result
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

Hashes

It is a collection of key-value pairs. It is also known as associative arrays. We use the “%” symbol in Perl to declare a hash. We utilize the “\$” symbol followed by the key in brackets to get to a specific value.

Example:

```
# Program to demonstrate the
# Hashes data type

# Hashes
%data = ('PFP', 7, 'for', 4, 'Peeks', 11);

#displaying result
print "\$data{'PFP'} = $data{'PFP'}\n";
print "\$data{'for'} = $data{'for'}\n";
print "\$data{'Peeks'} = $data{'Peeks'}\n";
```

This chapter covered Perl introduction, installation and environment setup and Hello World Program.

NOTE

1. Hello World Program in Perl.

Fundamentals of Perl

IN THIS CHAPTER

- Modes of Writing
- Boolean Values
- Operators
- Variables
- Modules in Perl
- Packages in Perl
- Number and Its Types
- Directories with CRUD Operations

In the previous chapter, we discussed Perl basic, and in this chapter, we will cover packages, module operators, variables, and modes of writing.

PERL CODE WRITING METHODS

Perl is a dynamic, high-level, interpreted, general-purpose programming language. Perl supports both procedural and object-oriented programming. Perl was initially designed only for system administration and text processing, but subsequent versions included the ability to manage regular expressions, network sockets, etc.

Perl is syntactically similar to other commonly known programming languages, making it easy to write and understand. Perl is a free-form

programming language, meaning it may be written, formatted, and indented according to the user's needs.

A Perl program consists of a series of statements, loops, subroutines, etc., which facilitates navigation around the code. Every Perl statement must conclude with a semicolon (;). Perl, like other languages, provides several writing and execution modes for Perl code. These modes may be classified according to their compatibility with writing and mode of execution in the following ways:

- Interactive mode
- Script mode
- One-liner mode

These modes can be used on the command line with the Perl keyword or in online Integrated Development Environment (IDEs) as a block of code. Along with the installation package, Perl has its own inbuilt IDE.

Interactive Mode

Interaction with the interpreter is signified by the interactive method of creating Perl code. Interactive mode is a wonderful approach to begin programming since it allows you to examine the flow of code line by line and simplifies the debugging process. With Perl debugger, interactive mode in Perl can be utilized on the command line.

This interpreter is often called REPL – Read, Evaluate, Print, Loop. Interactive mode enables the immediate creation and execution of code without creating a temporary file to save the source code. Using Perl debugger, the built-in Perl command line or the Windows command prompt may be used as a REPL. This debugger can be used with a Perl application with the following command:

```
perl -del
```

The user must write the code line by line in the interactive mode of writing Perl code, which is run simultaneously.

Perl's interactive mode may be run directly from the command line, without the need for a debugger. This can be done by using the following command:

```
perl -e Code_statement;
```

This statement utilizes the `-e` parameter to avoid script creation and allows the code to run without the debugger on the command line.

This way of writing in interactive mode does not allow the user to create multi-line code as it does in the debugger. This mode is not recommended for long programs.

Interactive mode is useful for teaching new programmers the fundamentals of programming, but it may become cumbersome and tedious when dealing with more than a few lines of code.

Script Mode

Script mode in Perl is used to develop Perl programs that include more than a few lines of code and are too complicated for interactive mode.

Using a text editor, script mode in Perl can be used to develop a Perl program; it is saved in a file called a script, and then the script file is run using the command line. This file must be saved with the `.pl` extension and stored in the same directory as the directory path specified on the command line. This script is then executed from the command line using the following command:

```
perl FileName.pl
```

Code is typed in a text editor (notepad, for example) and saved as a Perl program.`.pl` script.

Unlike the interactive mode, script mode in Perl cannot provide output for each individual expression.

In the interactive mode, the expression is evaluated, and the result is presented automatically, while in the script mode, the expression is evaluated, but the result is not displayed until prompted.

Online IDEs also provide script mode, which is used to develop and run Perl code without manually putting it in a file. In these IDEs, compiled code is saved in memory as a temporary file that is only useful when the code is run, and the browser containing the IDE is open. Once the page is refreshed, this temporary file is destroyed, and memory space is released. Online IDEs have simplified the execution of codes since they need less effort than the script mode, in which files must be saved in the machine's memory.

This quickens the compilation and execution of code. These online IDEs, although convenient for programmers, have significant restrictions, such as the inability to conduct file handling operations unless the file is

uploaded to their server, which poses a risk to sensitive data. Compilers that operate through the command line allow for such file manipulation capabilities.

Here is an example of a Perl program that adds two integers using an online IDE:

```
#!/usr/bin/perl
# add two numbers program

# Assigning values to the variables
$var1 = 20;
$var2 = 35;

# Evaluating result
$result = $var1 + $var2;

# Printing result
print "Result after the addition is: $result";
```

One-Liner Mode

Perl also has a one-liner mode, allowing us to type and run a minimal code script right from the command line. This is done to prevent creating files to store scripts for codes that are not too long. With the use of the following command, these codes can be written on a single line in command line mode:

```
perl -e
```

This command is used to write and execute one-liner code at the command line by enclosing it in double quotation marks. The `-e` flag in the preceding command informs the compiler that the code's script is not kept in any sort of file but is written in the double codes immediately after this flag.

These one-liners may be quite handy for making rapid modifications such as obtaining information and modifying file contents. Some programmers avoid using one-liners because they can become clumsy when the script becomes too long. While some programmers prefer doing this since one-liners are faster than scripts because they are not stored in files.

BOOLEAN VALUES IN PERL

True and False are considered boolean values in most computer languages. However, Perl does not provide the boolean type for True and False. When a function returns True or False, a programmer might use the word “boolean.” Scalar values, like conditional expressions (if, while, etc.), will return true or false.

Example:

```
# Perl Code to demonstrate boolean values

# variable assigned value 0
$x = 0;

# checking whether x is true or false
if ($x)
{
    print "x is True\n";
}
else
{
    print "x is False\n";
}

# variable assigned value 2
$y = 2;

# checking whether y is true or false
if ($y)
{
    print "y is True\n";
}
else
{
    print "y is False\n";
}
```

True values: In Perl, True values are any non-zero numbers other than zero. In Perl, string constants such as “true,” “false,” “00” (2 or more 0 characters), and “0n”(a zero followed by a newline character in the string) are also considered true values.

Example:

```
# Perl Code to demonstrate True values

# variable assigned value 5
$x = 5;

# checking whether x is true or false
if ($a)
{
    print "x is True\n";
}
else
{
    print "x is False\n";
}

# string variable assigned white
# space character
$y = ' ';

# checking whether y is true or false
if ($y)
{
    print "y is True\n";
}
else
{
    print "y is False\n";
}

# string variable assigned 'false'
# value to it
$m = 'false';

# checking whether c is true or false
if ($m)
{
    print "m is True\n";
}
else
{
    print "m is False\n";
}
```

```

}

# string variable assigned "0\n"
# value to it
$n = "0\n";

# checking whether d is true or false
if ($n)
{
    print "n is True\n";
}
else
{
    print "n is False\n";
}

```

False values: In Perl, false values are empty string or a stringcv containing a single digit 0 or undef value and zero.

Example:

```

# Perl Code to demonstrate False values

# variable assigned value 0
$x = 0;

# checking whether x is true or false
if ($x)
{
    print "a is True\n";
}
else
{
    print "x is False\n";
}

# string variable assigned empty string
$y = '';

# checking whether y is true or false
if ($y)

```

```

{
    print "y is True\n";
}
else
{
    print "y is False\n";
}

# string variable assigned undef
$m = undef;

# checking whether m is true or false
if ($m)
{
    print "m is True\n";
}
else
{
    print "m is False\n";
}

# string variable assigned ""
# value to it
$n = "";

# checking whether n is true or false
if ($n)
{
    print "n is True\n";
}
else
{
    print "n is False\n";
}

```

Note: For the conditional check, where the user must compare two distinct variables, if they are not equal, it returns False; otherwise, it returns True.

Example:

```

# Perl Program demonstrate conditional check

# variable initialized with string
$c = "PFP";

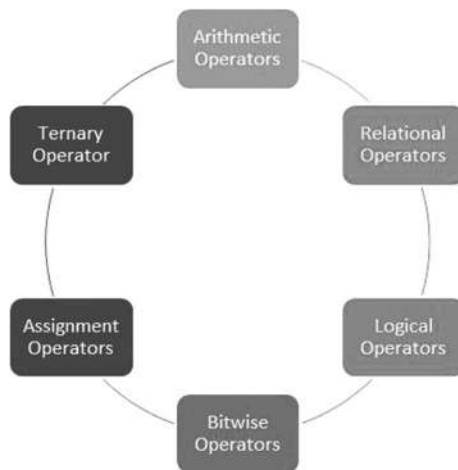
```

```
# using the if statement
if ($c eq "PFP")
{
    print "Return True\n";
}
else
{
    print "Return False\n";
}
```

OPERATORS

Operators are the building blocks of any computer language, allowing the programmer to perform various operations on operands. In Perl, the symbols for operators will differ depending on the kind of operand (like scalars and strings). Operators are classified based on their many functions¹:

- Arithmetic operators
- Relational operators
- Ternary operators
- Logical operators
- Bitwise operators
- Assignment operators



Types of operators.

Arithmetic Operators

There are various types of arithmetic operators.

Addition

These are used to execute arithmetic and mathematical operations on operands.

To add the values of the two operands, use the “+” operator. As an example:

```
$x = 15;  
$y = 20;  
print $x + $y;  
Here Result will be 25
```

Subtraction

The “-” operator subtracts the right-hand operand from the left-hand operand. As an example:

```
$x = 15;  
$y = 20;  
print $x - $y;  
Here Result will be 5
```

Multiplication

The “*” operator multiplies the values on both sides of the operator. As an example:

```
$x = 15;  
$y = 20;  
print $x * $y;  
Here Result will be 300
```

Division

When the first operand is divided by the second, the “/” operator returns the remainder. As an example:

```
$x = 40;  
$y = 20;  
print $x / $y;  
Here Result will be 20
```

Modulus

The modulus operator (%) divides the left-hand operand from the right-hand operand and returns the remainder. As an example:

```
$x = 10;
$y = 15;
print $x % $y;
Here Result will be 5
```

Exponent Operator

The “**” operator is used to calculate the exponential (power) of operands. As an example:

```
$x = 2;
$y = 3;
print $x ** $y;
Here Result will be 8
```

The aim of this program is to describe the following arithmetic operators:

```
# Perl Program to illustrate Arithmetic Operators

# Operands
$x = 10;
$y = 4;

# using arithmetic operators
print "The Addition is: ", $x + $y, "\n";
print "The Subtraction is: ", $x - $y, "\n" ;
print "The Multiplication is: ", $x * $y, "\n";
print "The Division is: ", $x / $y, "\n";
print "The Modulus is: ", $x % $y, "\n";
print "The Exponent is: ", $x ** $y, "\n";
```

Relational Operators

When comparing two values, relational operators are employed. These operators will either return 1 (true) or 0 (false). These operators are also known as the equality operators. These operators use various symbols to work on strings. We may use this to learn about the comparison operators operation on string.

- Equal to operator: “==” Determine whether two values are equal. If equals, return 1 else return nothing.
- Not equal to operator: “!=” Check to see whether the two values are equivalent. If the values are not equal, 1 is returned; else, nothing is returned.
- Comparison of equal to operator: “<=>” If the left operand is smaller than the right operand, the result is -1; otherwise, the result is 0.
- Greater than operator: “>” If the left operand is greater than the right operand, 1 is returned; else, nothing is returned.
- Less than operator: “<” If the left operand is greater than the right operand, 1 is returned; else, nothing is returned.
- Greater than equal to operator: “>=” If the left operand is larger than or equal to the right operand, 1 is returned; otherwise, nothing is returned.
- Less than equal to operator: “<=” If the left operand is smaller than or equal to the right operand, 1 is returned; otherwise, nothing is returned.

To demonstrate the relational operators in Perl, write a program:

```
# Perl Program to illustrate Relational Operators

# Operands
$x = 10;
$y = 60;

# using Relational Operators
if ($x == $y)
{
    print "Equal To Operator is True\n";
}
else
{
    print "Equal To Operator is False\n";
}

if ($x != $y)
```

```
{
print "Not Equal To Operator is True\n";
}
else
{
print "Not Equal To Operator is False\n";
}

if ($x > $y)
{
print "Greater Than Operator is True\n";
}
else
{
print "Greater Than Operator is False\n";
}

if ($x < $y)
{
print "Less Than Operator is True\n";
}
else
{
print "Less Than Operator is False\n";
}

if ($x >= $y)
{
print "Greater Than Equal To Operator is True\n";
}
else
{
print "Greater Than Equal To Operator is False\n";
}

if ($x <= $y)
{
print "Less Than Equal To Operator is True\n";
}
else
{
print "Less Than Equal To Operator is False\n";
}
```

```

}
if ($x <=> $y)
{
print "Comparison of Operator is True\n";
}
else
{
print "Comparison of Operator is False\n";
}

```

Logical Operators

These operators are being used to combine two or more conditions or to improve the original condition's evaluation.

- **Logical AND:** When both of the conditions in consideration are met, the “and” operator returns true. For instance, \$x and \$y are true when both x and y are true (i.e. non-zero). We can also use &&.
- **Logical OR:** When one of the conditions in consideration are met, the “or” operator returns true. For instance, \$x or \$y is true if either x or y is true (i.e. non-zero). Of course, it returns “true” if both x and y are true. We may also use ||.
- **Logical NOT:** If the condition under consideration is met, the “not” operator returns 1. For example, if \$z is 0, not(\$z) is true.

Program: To show how logical operators work:

```

# Perl Program to illustrate Logical Operators

# Operands
$x = true;
$y = false;

# AND operator
$result = $x && $y;
print "AND Operator: ", $result, "\n";

# OR operator
$result = $x || $y;
print "OR Operator: ", $result, "\n";

```

```
# NOT operator
$z = 0;
$result = not($z);
print "NOT Operator: ", $result;
```

Bitwise Operators

The bitwise operation is carried out using these operators. It will convert the integer to bits first, and then perform the bit-level action on the operands.

- & (bitwise AND) is a function which takes two operands and performs AND on each bit of the two numbers. AND returns only 1 if both bits are 1. As an illustration,

```
$x = 13;    // 1101
$y = 5;     // 0101
$z = $y & $x;
print $z;
Here Output will be 5
```

Explanation:

```
$x = 1 1 0 1
$y = 0 1 0 1
-----
$z = 0 1 0 1
-----
```

- | (bitwise OR) is a function that takes two operands and performs OR on each bit of two numbers. OR returns 1 if either of two bits is 1. As an illustration,

```
$x = 13;    // 1101
$y = 5;     // 0101
$z = $y | $x;
print $z;
Here Output will be 13
```

Explanation:

```
$x = 1 1 0 1
$y = 0 1 0 1
-----
$z = 1 1 0 1
-----
```

- ^ (bitwise XOR) takes two operands and performs XOR on each bit of the two numbers. If two bits are different, the result of XOR is 1. As an example:

```
$x = 13; // 1101
$y = 5;  // 0101
$z = $y ^ $x;
print $z;
Here Output will be 9
```

Explanation:

```
$x = 1 1 0 1
$y = 0 1 0 1
-----
$z = 1 0 0 1
-----
```

- ~ (Complement operator) is a unary operator that acts as a bit flipper. Its job is to reverse the bits and return the result in 2's complement form due to a signed binary number.
- (<<) Binary left shift operator accepts two integers, left shifts the bits of the first operand, and the second operand determines the number of places to shift. The left operand is multiplied by the number of times the right operand provides. As an example:

```
$x = 70;
$z = $x << 2;
print $z;
```

Output: 280

Explanation:

```
70 * 2 = 140 --- (1)
140 * 2 = 280 --- (2)
```

- (>>) Binary right shift operator takes two numbers, right shifts the bits of the first operand, and the second operand determines the number of places to shift. It divides the left operand by the number of times provided by the right operand. As an example:

```
$x = 80;
$z = $x >> 2;
```

```
print $z;
Output: 15
```

Explanation:

```
60 / 2    = 40    --- (1)
40 / 2    = 20    --- (2)
```

Program: To show how bitwise operators work:

```
# Perl Program to illustrate Bitwise operators
#!/usr/local/bin/perl
use integer;

# Operands
$x = 100;
$y = 2;

# Bitwise AND Operator
$result = $x & $y;
print "Bitwise AND: ", $result, "\n";

# Bitwise OR Operator
$result = $a | $b;
print "Bitwise OR: ", $result, "\n";

# Bitwise XOR Operator
$result = $x ^ $y;
print "Bitwise XOR: ", $result, "\n";

# Bitwise Complement Operator
$result = ~$x;
print "Bitwise Complement: ", $result, "\n";

# Bitwise Left Shift Operator
$result = $x << $y;
print "Bitwise Left Shift: ", $result, "\n";

# Bitwise Right Shift Operator
$result = $x >> $y;
print "Bitwise Right Shift: ", $result, "\n";
```

Assignment Operators

It is used to assign value to the variable. The assignment operator's left operand is a variable, while the assignment operator's right operand is a value.

The following are examples of assignment operators:

- “=” (Simple assignment): The most basic assignment operator. This operator assigns the variable on the left the value on the right.

As an example:

```
$x = 20;
$y = 40;
```

- “+=”(Add assignment): A combination of the “+” and “=” operators. This operator first adds current value of the variable on the left to value on the right, and then assigns result to the variable on the left.

As an example:

(`$x += $y`) can be written as (`$x = $x + $y`)

- If the initial value of `a` is 5, then (`$a += 6`) = 11.
- “-=”(Subtract assignment): This operator combines the “-” and “=” operators. This operator subtracts the variable’s current value from the value on right and assigns the resulting value to the variable on the left.

As an example:

(`$x -= $y`) can be written as (`$x = $x - $y`)

- If the initial value of `a` is 8, then (`$a -= 6`) = 2.
- “*=”(Multiply assignment): This operator combines the “*” and “=” operators. The variable on the left is multiplied by the value on the right, and the result is assigned to a variable on the left.

As an example:

(`$x *= $y`) can be written as (`$x = $x * $y`)

- If the initial value of `a` is 5, then (`$x *= 6`) = 30.
- “/=”(Division assignment): This operator combines the “/” and “=” operators. This operator divides the current value of the variable on the left by the value on the right and allocates the result to the variable on the left.

As an example:

(`$x /= $y`) can be written as (`$x = $x / $y`)

- If the initial value stored in `a` is 6, then (`$x /= 2`) = 3.

- “%=”(Modulus assignment): This operator combines the “%” and “=” operators. This operator modifies the current value of the variable on the left by the value on the right before assigning the result to the variable on the left.

As an example:

`($x %= $y)` can be written as `($x = $x % $y)`

- If the initial value stored in `a` is 6, then `($a percent = 2) = 0`.
- “**=”(Exponent assignment): This operator is a mixture of the “**” and “=” operators. This operator multiplies the current value of the variable on the left by the value on the right before assigning the result to the variable on the left.

As an example:

`($x **= $y)` can be written as `($x = $x ** $y)`

- If the initial value stored in `a` is 6, then `($a **= 2) = 36`.

Program: To illustrate how assignment operators function:

```
# program to demonstrate working of Assignment
Operators
#!/usr/local/bin/perl

# taking two variables & using
# the simple assignments operation
$x = 8;
$y = 5;

# using the Assignment Operators
print "The Addition Assignment Operator: ", $x +=
$y, "\n";

$x = 8;
$y = 4;
print "Subtraction Assignment Operator: ", $x -=
$y, "\n" ;

$x = 8;
$y = 4;
print "Multiplication Assignment Operator: ",
$x*=$y, "\n";
```

```

$x = 8;
$y = 4;
print "Division Assignment Operator: ", $x/=$y,
"\n";

$x = 8;
$y = 5;
print "Modulo Assignment Operator: ", $x%=$y, "\n";

$x = 8;
$y = 4;
print "Exponent Assignment Operator: ", $x**=$y,
"\n";

```

Ternary Operator

It is conditional operator, which is simple version of an if else statement. The term “ternary” refers to fact that it has three operands. Depending on the value of Boolean expression, it will return one of two values.

Syntax:

```
condition? firstexpression : secondexpression;
```

Example:

```

# Perl program to demonstrate working
# of the Ternary Operator

$c = 5;
$d = 10;

# To find which value is greater
# Using the Ternary Operator
$result = $c > $d? $c : $d;

# displaying the output
print "Larger Number is: $result"

```

Note: The condition in the ternary operator may also be any expression generated by utilizing other operators such as relational operators and logical operators.

```
# Perl program to demonstrate working
# of Ternary Operator by using the expression
# as a condition

# here maximum value can be 200
$MAX_VALUE = 200;

# suppose user provide value
$user_value = 666;

# To find which whether user provided
# value is satisfying maximum value
# or not by using the Ternary Operator
$result = $user_value <= $MAX_VALUE? $user_value :
$MAX_VALUE;

# displaying output
# Here it will be MAX_VALUE
print "$result"
```

VARIABLES IN PERL

Variables are used throughout Perl to store and manipulate data. A variable takes up memory space when it is created. A variable's data type assists the interpreter in allocating memory and deciding what to store in reserved memory. Variables can so hold integers, decimals, or texts by assigning multiple data types to the variables.²

Naming of a Variable

With the usage of a certain data type, a variable in Perl can be called anything. When naming a variable, there are various guidelines to follow:

- Perl variables are case-sensitive.

Example:

\$Seema and \$seema are two different variables

- According to the data type, it begins with \$, @, or % and is followed by zero or more letters, underscores, and digits.
- Variables in Perl are not permitted to contain white spaces or any special character other than underscore.

Example:

```
$my-name = "Seema"; // Invalid
$my name = "Seema"; // Invalid
$my_name = "Seema"; // Valid
```

Declaration of a Variable

The data type used to define the variable is used to declare the variable. These variables can have one of three data types:

- **Scalar variables:** Scalar variables comprise a single string or numeric value. It begins with the \$ symbol.

Syntax: \$varname = value;

Example:

```
$item = "Heyyy"
$item_one = 2
```

- **Array variables:** It comprises a set of values that have been randomly ordered. It begins with the @ symbol.

Syntax : @varname = (val1, val2, val3,);

Example:

```
@pricelist = (170, 60, 50);
@namelist = ("Grapes", "Apple", "Banana");
```

- **Hash variables:** It has (key, value) pairs that are efficiently accessible per key. It begins with the % sign.

Syntax : %varname = (key1=>val1, key2=>val2, key3=>val3,...);

Example:

```
%itempairs = ("Grapes" =>12, "Apple"=>5);
%pairrandom = ("Hello" =>7, "Bye"=>8);
```

Modification of a Variable

Perl allows us to change the values of variables after they've been declared. A variable can change in a variety of ways:

- A scalar variable's value can change simply by redefining it.

Example:

```
$name = "Seema";
# This can modify by simply
# redeclaring variable $name.
$name = "Sahil";
```

- An array element can update by passing its index to the array and assigning a new value to it.

Example:

```
@array = ("A", "B", "C", "D", "E");

# If the value of second variable is to
# modify then it can done by
$array[2] = "4";
# $array[2] = "4"; is alternate way of updating
the value in array.

# This will change array to,
# @array = ("A", "B", "4", "D", "E");
```

- A hash value can change by using its key.

Example:

```
%Hash = ("A", 20, "B", 30, "C", 40)
# This will modify the value
# assigned to Key 'B'
$Hash{"B"} = 56;
```

Variable Interpolation

Perl provides many ways for assigning a string to a variable. This may be accomplished by using single quotes, double quotes, the q-operator, the double-q operator, and so forth.

The use of single and double quotes for string writing is the same; however, there is a minor difference in how they operate. Strings enclosed in single quotes display the content contained within them exactly as typed.

Example:

```
$name = "Seema"
print 'Hello $name\nHow are you doing?'
```

Strings enclosed in double quotes, on the other hand, replace the variables with their values before displaying the string. It even replaces the escape sequences with their correct implementation.

Example:

```
$name = "Seema"
print "Hello $name\nHow are you doing?"
```

Example code:

```
#!/usr/bin/perl
use Data::Dumper;

# Scalar-Variable
$name = "PeeksForPeeks";

# Array Variable
@array = ("P", "E", "E", "K", "S");

# Hash Variable
%Hash = ('Welcome', 10, 'to', 20, 'Peeks', 40);

# Variable-Modification
@array[2] = "F";

print "The Modified Array is @array\n";

# Interpolation of a Variable

# Using the Single Quote
print 'Name is $name\n';

# Using the Double Quotes
print "\nName is $name";

# Printing the hash contents
print Dumper(\%Hash);
```

VARIABLES AND ITS TYPES

Variables are reserved memory locations for storing values. This means that creating a variable frees up memory space. The data type of a variable assists the interpreter in allocating memory and deciding what to store in

reserved memory. Variables can thus store integers, decimals, or strings by assigning different data types to the variables.

Perl has three basic data types, which are as follows:

- Scalars
- Arrays
- Hashes

As a result, Perl will employ three types of variables. A scalar variable, preceded by a dollar sign (\$), can store a number, a string, or a reference. An array variable stores ordered lists of scalars, denoted by the @ sign. The hash variable, preceded by sign %, will be used to store sets of key/value pairs.

Creating Variables

To reserve memory space, Perl variables do not need to be explicitly declared. Like in other computer languages, the operand to the left of the “=” operator is the variable’s name, and the operand to the right of the “=” operator is the value stored in the variable.

As an example:

```
$age = 30;
$name = "Sima";
$rollno = 32;
Here 30, "Sima" and 32 are the values assigned to
$age, $name and $roll no variables, respectively.
```

Scalar Variables

A scalar is a single data unit. The data could be an integer number, a floating-point number, a character, a string, a paragraph, or an entire web page. Here’s an example of how to use scalar variables:

```
#!/usr/bin/perl

# Assigning the values to scalar
# variables
$age = 30;
$name = "Sima";
$rollno = 32;
```

```
# Printing the variable values
print "Age = $age\n";
print "Name = $name\n";
print "Roll no = $rollno\n";
```

Array Variables

An array variable stores ordered list of scalar values. Array data type variables are preceded by “at” (@) sign. The dollar sign (\$) refers to a single array element with the variable name followed by the element’s index in square brackets.

Here’s an example of using an array variable:

```
#!/usr/bin/perl

# Assigning the values to Array variables
@ages = (43, 70, 33); # @ is used to declare
                        # the array variables
@names = ("ABC", "DEF", "GHI");

# Printing values of Arrays
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

We used “\” before the “\$” sign to print it as a statement. Otherwise, Perl will interpret it as a variable and print the value stored in it.

Hash Variables

Hash is a collection of the key-value pairs. Hash variables are preceded by a modulus (%) sign. In the hash, keys are used to refer to a single variable. In curly brackets, the hash variable name is followed by the key associated with the value to access these elements.

Here is a simple example of a Hash variable:

```
#!/usr/bin/perl

# Defining Hash variable using '%'
%data = ('ABC', 55, 'DEF', 80, 'GHI', 44);
```

```
# Printing values of Hash variables
print "\$data{'XYZ'} = $data{'ABC'}\n";
print "\$data{'LGH'} = $data{'DEF'}\n";
print "\$data{'KMR'} = $data{'GHI'}\n";
```

Variable Context

Perl treats the same variable differently depending on the context, i.e. the situation in which a variable is used.

Example:

```
#!/usr/bin/perl

# Defining Array variable
@names = ('ABC', 'DEF', 'GHI');

# Assigning values of the array variable
# to another array variable
@copy = @names;

# Assigning values of the Array variable
# to a scalar variable
$size = @names;

# Printing values of new variables.
print "The Given names are : @copy\n";
print "The Number of names are : $size\n";
```

@names is an array that has been used two times in this context. First, we copied it into other array, i.e. a list, so that it would return all the elements if the context was a list context. The same array is then attempted to be stored in a scalar context, which by default returns just the number of elements in this array. The table below depicts the various contexts:

S. no.	Context and description
1.	Scalar In a scalar context, assignment to a scalar variable evaluates the value to the right of “=”.
2.	List In a list context, assignment to an array or a hash evaluates the right of “=”.

(Continued)

S. no.	Context and description
3.	Boolean A boolean context is a location where an expression is evaluated to see whether it is true or false.
4.	Void This context does not care what the return value is, and it doesn't even want one.
5.	Interpolative This context will only appear within quotation marks (""") or with things that behave like quotation marks (""").

SCOPE OF VARIABLES

The scope of the variable is the area of program where the variable is accessible. The visibility of variables in a program is also referred to as a scope. We can declare global variables or private variables in Perl. Lexical variables are another name for private variables.

The Scope of Global Variables

Global variables can be used within any function or block that has been created within the program. It is visible throughout the program. Global variables can be used directly and are accessible from anywhere in the program.

First example: At the start of the code, the variable \$name is declared. It will be visible until the end of the file. Even within blocks. Even if they are included in the function declarations. If we change the variable within the block, the value for the rest of the code will change, even outside the block.

```
#!/usr/bin/perl

# Defining Array variable
@names = ('ABC', 'DEF', 'GHI');

# Assigning values of the array variable
# to another array variable
@copy = @names;

# Assigning values of the Array variable
# to a scalar variable
$size = @names;
```

```

# Printing values of new variables.
print "Given names are : @copy\n";
print "Number of names are : $size\n";
# Perl program to illustrate
# Scope of Global variables

# declaration of the global variable
$name = "PFP";

# printing global variable
print "$name\n";

# global variable can use
# inside a block, hence the we
# are taking a block in which
# we will print the value of
# $name i.e. global variable
{

    # here GFG will print
    print "$name\n";

    # values in the global variable can be
    # changed even within block,
    # hence value of $name is
    # now changed to "PeeksforPeeks"
    $name = "PeeksforPeeks";

    # print function prints
    # "PeeksforPeeks"
    print "$name\n";
}

# changes made inside the above block'
# are reflected in the whole program
# so here PeeksforPeeks will print
print "$name\n";

```

Second example:

```

# program to illustrate the
# Scope of Global variables

```

```

# declaration of the global variables
$name = "PFP";
$count = 1;

# printing global variables
print $count." ".$name."\n";
$count++;

# Block starting
{

    # global variable can use inside
    # a block, so below statement will
    # print PFG and 1
    print $count." ".$name."\n";

    # incrementing the value of
    # count inside block
    $count++;
}

# taking function
sub func {

    # Global variable, $count and $name,
    # are accessible within the function
    print $count." ".$name."\n";
}

# calling function
func();

```

Lexical Variables' Scope (Private Variables)

Private variables in Perl are defined by prefixing the variable with “my” keyword. The “my” keyword limits variables inside a specified function or block. A block can either be a for loop, a while loop, or a block of code surrounded by curly braces.

The local variable’s scope is local; it exists just between these two curly brackets (block of code); this variable does not exist outside of this block. These variables are sometimes referred to as lexical variables. When private variables are used inside a function or block, the global variables with the same name are hidden. When a subroutine is called with a private

variable, that variable may utilize inside the procedure. When the procedure terminates, the private variables are no longer usable.

Example:

```
# program to illustrate the
# scope of private variables

# declaration of the global variable
$name = "Global";
$count = 1;

# printing the global variables
print $count." ".$name."\n";

# incrementing the value of count
# i.e it become 2
$count++;

# block starting
{

    # declaring private variable by using my
    # keyword which can only use
    # within this block
    my $new_name = "Private";

    # global variables are
    # accessible inside the block
    print $count." ".$name."\n";

    # incrementing the value
    # of global variable
    # here it become 3
    $count++;

    print $name." and ".$new_name."\n";
}

# $new_name variable cannot
# be used outside, hence nothing
# is going to print
```

```

print "Variable defined in the above block:
".$new_name."\n";

# declaring function
sub func {

    # this private variable declaration
    # hides global variable which define
    # in the beginning of program
    my $name = "Hide";
    print $count." ".$name."\n";

}

# calling function
func();

```

Package Variables

Package scoping is an additional type of scoping in Perl. This is used when we need to create variables that can only be used in different namespaces. In every Perl program, the default namespace is “main.” The package keyword is used in Perl to define namespaces.

Example:

```

# program to illustrate
# Package Variables

# variable declared in the
# main namespace
$var1 = "Main Namespace";

print "The Value of Var1: ".$var1."\n";

# package declaration
# Pack1 is package
package Pack1;

    # since $var1 belongs to main namespace,
    # so nothing will print inside the Pack1
    # namespace
    print "The Value of var1: ".$var1."\n";

```

```

# variable declared in Pack1 namespace
# having same name as the main namespace
$var1 = "Pack1 Namespace";

# here $var1 belongs to Pack1 namespace
print "The Value of var1: ".$var1."\n";

# in-order to print variables
# from both namespace, use
# the following method
print "The Value of var1: ".$main::var1."\n";
print "The Value of var1: ".$Pack1::var1."\n";

```

In Perl, the “our” keyword only creates an alias for an existing package variable with the same name. Only within lexical scope of the “our” declaration can a package variable be used without qualifying it with the package name. A variable declared with our keyword is an alias for a package variable visible throughout its entire lexical scope, including across package boundaries.

```

# program to illustrate the use
# of our keyword

# Pack1 namespace declared
# by using package keyword
package Pack1;

    # declaring $Pack1::first_name
    # for the rest of lexical scope
    our $first_name;
    $first_name = "Shashank";

    # declaring $Pack1::second_name for the
    # only this namespace
    $second_name;
    $second_name = "Sharma";

# Pack2 namespace declared
package Pack2;

    # prints value of $first_name, as it
    # refers to $Pack1::first_name

```

```
print "first_name = ".$first_name."\n";

# It will print nothing as the $second_name
# doesn't exist in Pack2 package scope
print "second_name = ".$second_name."\n";
```

MODULES IN PERL

In Perl, a module is a group of connected subroutines and variables that execute a set of programming tasks. Perl modules can reuse. The comprehensive Perl archive network (CPAN) hosts several Perl modules. These modules are divided into several categories, including network, CGI, XML processing, and database interface.

Making a Perl Module

A module's name must be the same as the Package's name and conclude with the .pm extension.

Example:

```
package Calculator;

# Defining sub-routine for Multiplication
sub multiplication
{
    # Initializing Variables x & y
    $x = $_[0];
    $y = $_[1];

    # Performing operation
    $x = $x * $y;

    # Function to print Sum
    print "\n***Multiplication is $x";
}

# Defining sub-routine for the Division
sub division
{
    # Initializing Variables x & y
    $x = $_[0];
    $y = $_[1];
```

```

    # Performing operation
    $x = $x / $y;

    # Function to print answer
    print "\n***Division is $x";
}
1;

```

The file is called “Calculator.pm” and is located in the directory Calculator. To return true value to the interpreter, 1 is placed at the end of the function. Instead of 1, Perl accepts anything true.

Importing and Using a Perl Module

We utilize need or use functions to import this calculator module. :: is used to access a function or variable from a module. Here’s an example to show:

```

#!/usr/bin/perl

# Using Package 'Calculator'
use Calculator;

print "Enter the two numbers to multiply";

# Defining values to the variables
$x = 15;
$y = 20;

# Subroutine-call
Calculator::multiplication($x, $y);

print "\nEnter the two numbers to divide";

# Defining values to variables
$x = 55;
$y = 65;

# Subroutine call
Calculator::division($x, $y);

```

Utilizing Module Variables

Declaring variables from various packages before using them allows them to be utilized.

Example:

```
#!/usr/bin/perl

package Message;

# Variable-Creation
$username;

# Defining-subroutine
sub Hello
{
    print "Hello $username\n";
}
1;
```

The module's Perl file is as follows:

```
#!/usr/bin/perl

# Using the Message.pm package
use Message;

# Defining the value to variable
$message::username = "Peeks";

# Subroutine-call
Message::Hello();
```

Making Use of predefined Modules

Perl has several predefined modules that can be used in Perl applications at any time.

Such as “strict” and “warnings”.

Example:

```
#!/usr/bin/perl

use strict;
use warnings;

print " Hello This program uses the pre-defined
Modules";
```

PERL PACKAGES

A Perl package is a set of code that resides in its namespace. A Perl module is a package defined in a file with the same name as the package and the extension `.pm`. A variable or function with same name may exist in two distinct modules. Any variable that isn't in a package belongs to the main package. As a result, all variables used are part of the "main" package. The declaration of extra packages ensures that variables from various packages do not interfere with one another.

Perl Module Declaration

The module's name must be the same as the package name and has a `.pm` extension.

Example:

```
package Calculator;

# Defining sub-routine for the Addition
sub addition
{
    # Initializing Variables x & y
    $x = $_[0];
    $y = $_[1];

    # Performing the operation
    $x = $x + $y;

    # Function to print Sum
    print "\n***Addition is $x";
}

# Defining sub-routine for the Subtraction
sub subtraction
{
    # Initializing Variables x & y
    $x = $_[0];
    $y = $_[1];

    # Performing the operation
    $x = $x - $y;
```

```

    # Function to print difference
    print "\n***Subtraction is $x";
}
1;

```

The file is called “Calculator.pm” and is stored in the directory Calculator. To return true value to the interpreter, 1 is written after the function. Instead of 1, Perl accepts anything that is true.

Making Use of a Perl Module

We utilize need or use functions to access this calculator module. :: is used to access a function or variable from a module. Here’s an example to show:

```

#!/usr/bin/perl

# Using Package 'Calculator'
use Calculator;

print "Enter the two numbers to add";

# Defining values to variables
$x = 10;
$y = 20;

# Subroutine-call
Calculator::addition($x, $y);

print "\nEnter the two numbers to subtract";

# Defining values to the variables
$x = 30;
$y = 10;

# Subroutine-call
Calculator::subtraction($x, $y);

```

Using a Different Directory to Access a Package

If a file accessing the package is located outside the directory, we use “::” to specify the module’s path. For example, because a file that uses the calculator module is outside the calculator package, we write Calculator :: Calculator for loading the module, where the value on the left of the “::”

represents the package name and the value on the right of the “::” represents the Perl module name. To illustrate, consider the following example:

```
#!/usr/bin/perl

use GFG::Calculator; # Directory_name::module_name

print "Enter the two numbers to add";

# Defining values to variables
$x = 10;
$y = 20;

# Subroutine-call
Calculator::addition($x, $y);

print "\nEnter the two numbers to subtract";

# Defining values to variables
$x = 30;
$y = 10;

# Subroutine-call
Calculator::subtraction($x, $y);
```

Utilizing Module Variables

Declaring variables from various packages before using them allows them to be utilized. The following example exemplifies this.

```
#!/usr/bin/perl

package Message;

# Variable-Creation
$username;

# Defining-subroutine
sub Hello
{
print "Hello $username\n";
}
1;
```

The module's Perl file is as follows.

```
#!/usr/bin/perl

# Using the Message.pm package
use Message;

# Defining value to the variable
$Message::username = "XYZ";

# Subroutine-call
Message::Hello();
```

Begin and End Block

When we wish to run some code at the beginning and some code at the conclusion, we utilize the BEGIN and END blocks. The codes within BEGIN... are run at the beginning of the script, while the codes within END... are executed at the end. This is demonstrated in the following program:

```
#!/usr/bin/perl

# Predefined BEGIN block
BEGIN
{
    print "In begin block\n";
}

# Predefined END block
END
{
    print "In end block\n";
}

print "Hello Everyone;\n";
```

NUMBER AND ITS TYPES IN PERL

In Perl, a number is a mathematical object that may use to count, measure, and perform different mathematical operations. A numeral is a notational sign that symbolizes a number. In addition to being used in mathematical operations, these numerals are also utilized for ranking (in the form of serial numbers).

Example:

2, 4, -6, 7.6, -8.9, 057, 1.87e-10, 0xFFFF

These numbers can be classified into several sorts based on their application:

- **Integers:** Perl integers are represented as decimal numbers with a base of 10. These numbers might be both positive and negative. On the other hand, Perl employs “_” to represent large integer values to improve readability.

Example: 213,254,484 is represented as 213_254_484
#!/usr/bin/perl

```
# Positive-Integer
$c = 30;
```

```
# Negative-Integer
$d = -25;
```

```
# Big Integer-Number
$e = 213_254_484;
```

```
# Printing these numbers
print("Positive Integer: ", $c, "\n");
```

```
print("Negative Integer: ", $d, "\n");
```

```
print("Big Integer: ", $e, "\n");
```

- **Floating point numbers:** In Perl, floating-point numbers have an integer and fractional values separated by a decimal point. The integer component of a floating number can be positive or negative, while the fractional component can only be positive. In Perl, floating numbers may be expressed in two ways:
 - **Fixed point:** The decimal point is fixed in this format. This decimal point marks the beginning of the fractional portion.

Example:

22.5978

- **Scientific:** This representation comprises two parts: the significand (the actual number) and the exponent (the power of 10 by which the significand is multiplied).

Example:

```
1.23567e-3 represents 1235.67
#!/usr/bin/perl

# Positive Floating-Number
$c = 20.5647;

# Negative Floating-Number
$d = -15.2451;

# Scientific-value
$e = 123.5e-10;

# Printing these numbers
print("Positive Number: ", $c, "\n");

print("Negative Number: ", $d, "\n");

print("Scientific Number: ", $e, "\n");
```

- **Hexadecimal numbers:** Hexadecimal numerals have a base of 16, ranging from 0 to 15, and are written as 0xa with “0x” before the number, whereas “a” here represents the value of 10 in Hex form. These alphabets represent 10–15 and range from “a” to “f”. Hexadecimal numerals can have both positive and negative values.

Example:

```
0xe represents 14 in the Hex, and 0xc represents
12
#!/usr/bin/perl

# Positive Hexadecimal Number
$c = 0xc;

# Negative Hexadecimal Number
$d = -0xe;
```

```
# Printing these values
print("Positive Hex Number: ", $c, "\n");
print("Negative Hex Number: ", $d, "\n");

# To print Hex value
printf("Value in the Hex Format: %c", $c);
```

As a result, “% x” is used to display the value of the number in hexadecimal format, as illustrated above.

- **Octal numbers:** Octal numbers are numbers with a base of 8, ranging from 0 to 7. These numbers are represented as 057, with the number preceded by a “0” and the remainder being the octal value of the requisite decimal number. Octal numbers, like other sorts, can be both positive and negative.

Example:

For octal number 057 decimal equivalent will be 47.

```
#!/usr/bin/perl
```

```
# Positive Octal Number
$c = 074;
```

```
# Negative Octal Number
$d = -074;
```

```
print("The Positive Octal number: ", $c, "\n");
print("The Negative Octal number: ", $d, "\n");
```

```
# To print value in the Octal Form
printf("Value in Octal Form: %o", $c);
```

“% o” is used here to output the value in the octal form.

- **Binary numbers:** Binary numbers have a base of two, meaning they have only two values: 0 and 1. These numbers are written as 0b1010, with the letter “0b” before the number.

Example:

For binary number '0b1010' decimal equivalent will be '10'

```
#!/usr/bin/perl

# Positive Binary-Number
$c = 0b1010;

# Negative Binary-Number
$d = -0b10110;

# Printing these values
print("The Positive Binary Number: ", $c, "\n");
print("The Negative Binary Number: ", $d, "\n");

# Printing in the unsigned binary form
printf("Value in unsigned Binary Form: %b", $c);
```

“%b” is used in the above code to display the number in its binary form.

DIRECTORIES WITH CRUD OPERATIONS IN PERL

Perl is a universal and cross-platform programming language that is mainly used for text manipulation and is utilized in developing several software applications such as web development and graphical user interface applications. It is favored over other programming languages because it is quicker and more powerful, and it also includes many shortcuts that aid in creating rapid scripts.

In computer languages, a directory is used to store values in the form of lists. A directory is comparable to a file in many ways. The directory, like a file, may be used to conduct various activities. These activities are used to modify an existing directory or create a new one.

The following procedures can be done on a directory:

- Making a new directory.
- Opening an existing directory.
- Reading directory content.
- Modifying a directory path.
- Directory closing.
- Delete a directory.

Making a New Directory

`mkdir(PATH, MODE)` is used to create a directory. This function assists in creating a new directory; if the user wishes to check whether the file already exists, the `-e` function can be used. `PATH` sets the path using the mode specified by the `MODE` function.

Example:

```
#!/usr/bin/perl

# Directory Path
my $directory = 'C:\Users\PeeksForPeeks\Folder\
Perl';

# Creating directory in perl
mkdir($directory) or die "No $directory directory,
$!";
print "Directory created \n";
```

Opening an Existing Directory

In Perl, the short function `opendir DIRHANDLE, PATH` is used to open a directory. The `PATH` parameter defines the path to the directory to be opened.

Example:

```
#!/usr/bin/perl

my $directory = 'C:\Users\PeeksForPeeks\Folder';
opendir (DIR, $directory) or die "No directory,
$!";
while ($file = readdir DIR)
{
    print "$file\n";
}
closedir DIR;
```

Read Directory in the Scalar and List Context

Reading a directory is a regular operation since one must read what is stored in the files every time the code is run or understood. `readdir DIRHANDLE` is used to read a directory. A user can read the directory in two ways: list context and scalar context.

In the list context, the code returns all the remaining entries in the directory. If the entries in the directories are empty, the undefined values in the scalar context and the empty list in the list context are returned.

Scalar context:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

# Directory Path
my $directory = shift // 'C:\Users\GeeksForGeeks\
Folder';

# Opening directory
opendir my $dh, $directory or
die "Could not open '$directory' for the reading
'$!'\n";

# Printing content of directory
while (my $content = readdir $dh)
{
    say $content;
}

# Closing directory
closedir $dh;
```

List context:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

# Directory Path
my $directory = shift // 'C:\Users\PeeksForPeeks\
Folder';

# Opening-directory
opendir my $dh, $directory or
die "Could not open '$directory' for the reading
'$!'\n";
```

```
# Reading content of file
my @content = readdir $dh;

# Printing content of the directory
foreach my $content (@content)
{
    say $content;
}

# Closing directory
closedir $dh;
```

Modifying Directory Path

The `chdir()` method is used to modify a directory. This function assists in changing the directory and sending it to a different place. When invoked with a script, the `chdir()` function changes the directory for the rest of the script. The directory on the terminal will not be updated if this function is called from within a script. However, when called directly with a new directory path, differences are displayed in the terminal at the same time.

Example: When the function `chdir()` is used within a script:

```
#!/usr/bin/perl

# Module to return
# the current directory path
use Cwd;

# Path of the new directory
$directory = "C:/Users";

# Printing the path of the current directory
# using cwd function
print "Current directory is ";
print(cwd);
print "\n";

# Changing directory using the chdir function
chdir($directory) or
    die "Couldn't go inside $directory directory,
$!";

# Printing path of changed directory
print "Directory has change to $directory\n";
print "Current directory is ";
```

```
print(cwd);
print "\n";
```

Directory Closing

Closedir DIRHANDLE is used to shut a directory. DIRHANDLE is the handle of the directory that is opened using the opendir function.

Example:

```
#!/usr/bin/perl

# Directory which is to be opened
$dirname = "C:/Users/PeeksForPeeks";

# Opening directory
# using opendir function
opendir (dir, $dirname) || die "Error $dirname\n";

# Printing content of directory
# using the readdir function
while(($filename = readdir(dir)))
{
    print("$filename\n");
}

# Closing directory using closedir
closedir(dir);
```

Delete a Directory

The rmdir function can be used to remove a directory. This function removes the supplied directory by FILENAME only if it is empty; if successful, it returns true; otherwise, it returns false.

Example:

```
#!/usr/bin/perl
$directory = "C:/Users/PeeksForPeeks/Folder/Perl";

# This removes the Perl directory
rmdir($directory) or
    die "Couldn't remove $directory directory, $!";

print "Directory-removed \n";
```

In this chapter, we covered the fundamental of Perl, where we discussed modes of writing, Boolean values, operators, and variables. Furthermore, we covered modules in Perl, packages in Perl, numbers and their types, and directories with CRUD operations.

NOTES

1. Perl - Operators.
2. Perl | Variables.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Input and Output in Perl

IN THIS CHAPTER

- Use of `print()` and `say()`
- `print` Operator
- Use of `STDIN` for Input

In the previous chapter, we covered the Fundamental of Perl, and in this chapter, we will cover input and output with its relevant examples.

PERL `print()` AND `say()` METHODS

Perl evaluates input provided by the user or presented as Hardcoded input in the code using statements and expressions. Because it has been evaluated in the compiler, this evaluated expression will not display to the programmer. Perl employs the `print()` and `say()` functions to display the evaluated expression. These routines can show whatever parameters are supplied to them.

`print()` Operator

The `print()` operator in Perl is used to print the values of expressions in a List supplied to it as an input. The `print` operator returns anything passed to it as an argument, whether it's a string, a number, a variable, or anything else. This operator's delimiter in double quotes (`"`).

Syntax:

```
print "";
```

Example:

```
#!/usr/bin/perl -w

# Defining string
$string1 = "Peeks For Peeks";
$string2 = "Welcome all";

print "$string1";
print "$string2";
```

The above example uses the `print` function to print both strings, but they are written on the same line. To avoid this and print them on distinct lines, we must use the “`\n`” operator, which changes the line whenever it is used.

Example:

```
#!/usr/bin/perl -w

# Defining a string
$string1 = "Peeks For Peeks";
$string2 = "Welcome all";

print "$string1\n";
print "$string2";
```

If we use single quotes instead of the double quotes, the `print()` method will not output the values of the variables or the escape characters used in the statement, such as “`\n`”. These characters will be printed in their entirety and will not be evaluated.

```
#!/usr/bin/perl -w

# Defining string
$string1 = 'Peeks For Peeks';
$string2 = 'Welcome all';

print '$string1\n';
print '$string2';
```

say() Function

The `say()` method in Perl functions similarly to the `print()` function, with one minor difference: the `say()` function automatically inserts a newline at the end of the statement, removing the need to use the newline character “`\n`” to change the line.

Example:

```
#!/usr/bin/perl -w
use 5.010;

# Defining string
$string1 = "Peeks For Peeks";
$string2 = "Welcome all";

# say() function to print
say("$string1");
say("$string2");
```

We used “`use 5.010`” to utilize the `say()` function since newer versions of Perl don’t support some of the earlier versions’ features; thus, the older version is used to run the `say()` method.

print OPERATOR

In Perl, the `print` operator is used to print the values of expressions in a List that is passed to it as an argument. The `print` operator returns anything passed to it as an argument, whether it’s a string, a number, a variable, or anything else. This operator is delimited by double quotes (“”).

Syntax:

```
print ""
Returns:
0 on failure and 1 on success.
```

First example:

```
#!/usr/bin/perl -w

# Defining a string
```

```

$string = "Geeks For Geeks";

# Defining an array of Integers
@array = (20, 30, 40, 50, 60, 70);

# Searching a pattern in string
# using index() function
$index = index ($string, 'or');

# Printing the position of the matched pattern
print "Position of 'or' in string $index\n";

# Printing defined array
print "Array of the Integers is @array\n";

```

Second example:

```

#!/usr/bin/perl -w

# Defining string
$string = "Welcome to PFP";

# Defining an array of integers
@array = (-20, 30, 25, -50, 55, -70);

# Searching a pattern in string
# using index() function
$index = index($string, 'o P');

# Printing the position of the matched pattern
print "Position of 'o P' in string $index\n";

# Printing the defined array
print "Array of the Integers @array\n";

```

USE OF STDIN FOR INPUT

Perl allows the programmer to receive user input and conduct actions on it. This allows the user to enter input other than the one provided by the programmer as Hardcoded input. The print() method can then be used to parse and print this input.

<STDIN> can provide keyboard input to a Perl application. STDIN is an abbreviation for Standard Input. There is no need to include STDIN

between the “diamond” and “spaceship” operators, i.e. `<>`. This is standard procedure. The `<>` operator can also be used to write to files. `<STDIN>` can be used in both Scalar and List contexts.

Syntax:

```
$a = <STDIN>; or $a = <>;
```

Example:

```
#!/usr/bin/perl -w
use strict;
use warnings;

print "Enter text:";
my $string = <STDIN>;

print "We entered $string as a String";
```

After entering Input, the above code requires us to press ENTER. This ENTER instructs the compiler to run the following line of code. However, `<STDIN>` treats the ENTER key as part of the input and prints the line when we hit it. Following the Input string, a new line will be written automatically. To avoid this, the `chomp()` method is used. This method will delete the newline character placed at the end of the user-supplied Input.

Example:

```
#!/usr/bin/perl -w
use strict;
use warnings;

print "Enter text:";
my $string = <STDIN>;
chomp $string;

print "We entered $string as a String";
```

In this chapter, we covered the use of `print()` and `say()`, print operator, and `STDIN` for Input.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Control Flow in Perl

IN THIS CHAPTER

- Decision-Making
- Loops
- when Statement
- goto Operator
- next Operator
- redo Operator

In the previous chapter, we discussed input and output in Perl, and in this chapter, we will cover control flow statements.

DECISION-MAKING IN PERL

Making decisions in programming is analogous to making decisions in real life. When a given condition is met in programming, a particular code block must be performed. Control statements are used in a programming language to control program execution flow based on particular criteria. These affect the execution flow to progress and branch based on changes in a program's state.

Perl decision-making statements:

- if
- if else

- nested if
- if elsif ladder
- unless
- unless else
- unless elsif

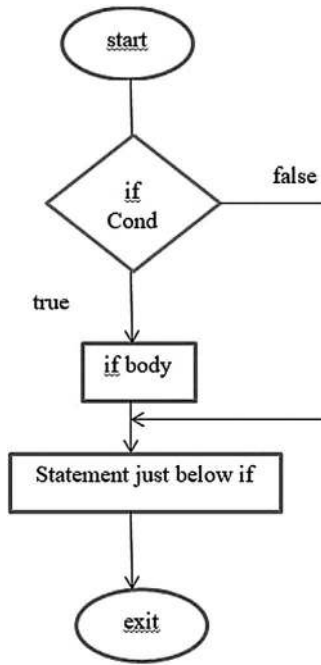
if Statement

The if statement is similar to those seen in other programming languages. It is used to carry out simple condition-based tasks. It is used to determine whether a specific statement or block of statements will be executed; for example, if a given condition is true, then a block of statements is executed; otherwise, it is not.

Syntax:

```
if (condition)
{
    # code executed
}
```

There will be a build time error if curly brackets {} are not utilized with if statements. As a result, brackets {} must be used with the if statement.

Flowchart:

Statement of if.

Example:

program to illustrate the if statement

```
$c = 10;
```

```
# if condition to check
```

```
# for the even number
```

```
if($c % 2 == 0 )
```

```
{
```

```
    printf "Even-Number";
```

```
}
```

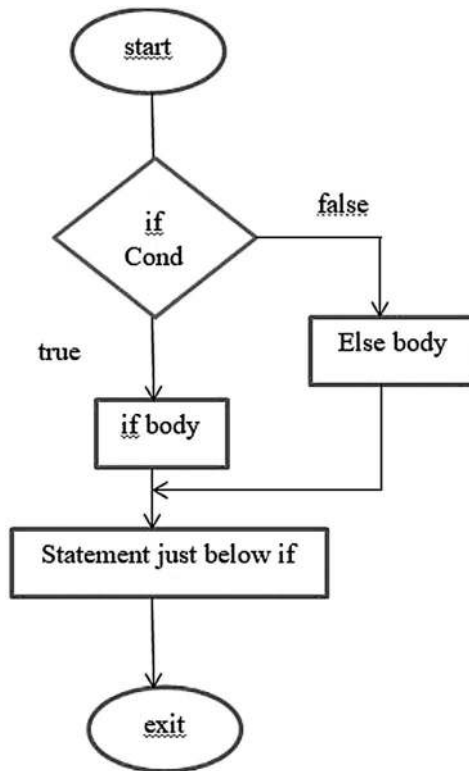
if else Statement

If the condition is true, the if statement evaluates the code; if the condition is false, the else statement is used. It instructs the code on what to do if the condition is false.

Syntax:

```
if(condition)
{
    # if a condition is true
}
else
{
    # if a condition is false
}
```

Flowchart:



Statement of if else.

Example:

```
# program to illustrate
# if else statement

$c = 21;

# if condition to check
# for the even number
if($c % 2 == 0 )
{
    printf "Even-Number";
}
else
{
    printf "Odd-Number\n";
}
```

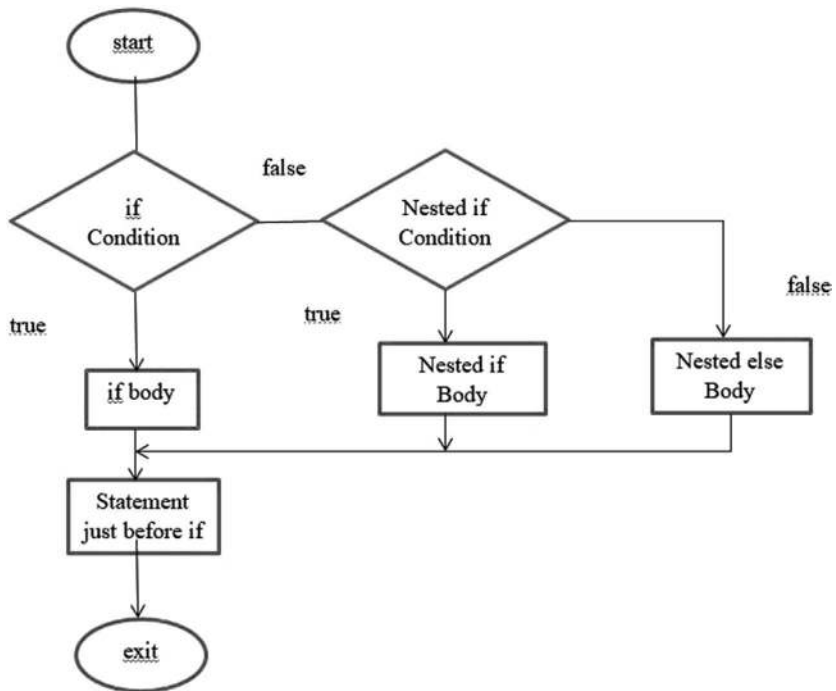
Nested if Statement

Nested if is an if statement inside an if statement. In this situation, the if statement is the target of another if or else statement. Nested if can be used when more than one condition must be true and one of the conditions is a sub-condition of the parent condition.

Syntax:

```
if (condition1)
{
    # Executes when the condition1 is true

    if (condition2)
    {
        # Executes when the condition2 is true
    }
}
```

Flowchart:

Statement of nested if.

Example:

```

# program to illustrate
# the Nested if statement

$c = 10;

if($c % 2 ==0)
{
    # Nested if statement
    # Will only executed
    # if above if statement
    # is true
    if($c % 5 == 0)
    {

```

```

        printf "The Number is divisible by 2 and
5\n";
    }
}

```

if elsif else ladder Statement

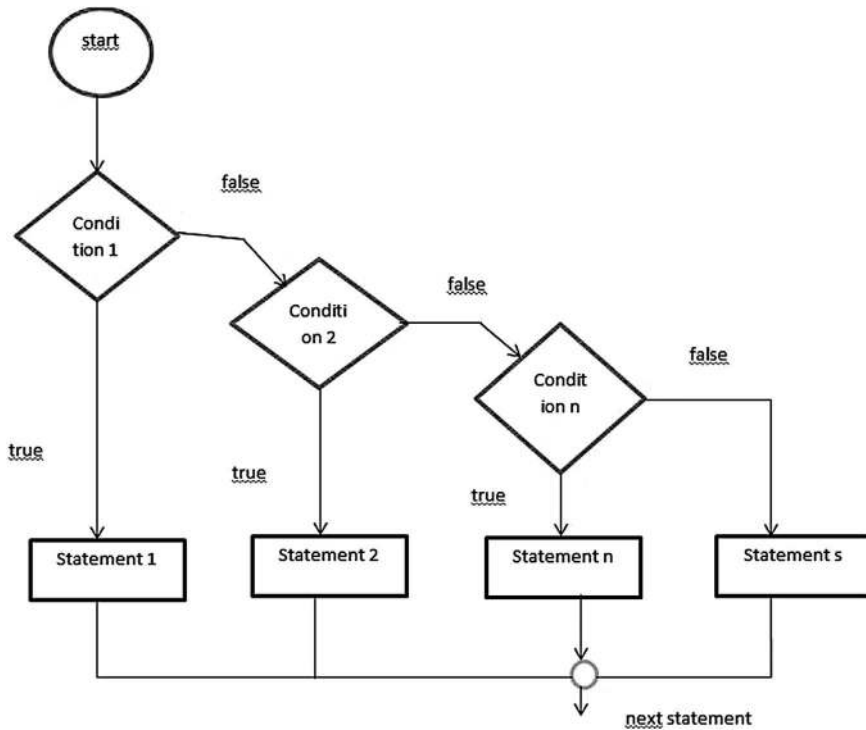
A user can select from a number of alternatives here. The if statements are performed in the order listed. When one of the if conditions is met, the statement associated with that condition is performed, and rest of the ladder is skipped. If none of the conditions are met, the last else expression is used.¹

Syntax:

```

if(condition1)
{
    # If condition 1 is true, code will be
    executed.
}
elseif(condition2)
{
    # code to be executed if the condition2
    is true
}
elseif(condition3)
{
    # code to be executed if the condition3
    is true
}
....
else
{
    # code to be executed if all conditions
    are false
}

```

Flowchart:

Statement of if else if.

Example:

```
# Perl program to illustrate
# if - elsif ladder statement
```

```
$c = 30;
```

```
if($c == 20)
{
    printf "c is 20\n";
}
```

```
elsif($c == 25)
{
    printf "c is 25\n";
}
```

```

elseif($c == 30)
{
    printf "c is 30\n";
}

else
{
    printf "c is not present\n";
}

```

unless Statement

If condition is false, the statements will execute. In a boolean context, the number 0, the empty string "", the character "0", the empty list (), and undef are all false, whereas all other values are true.

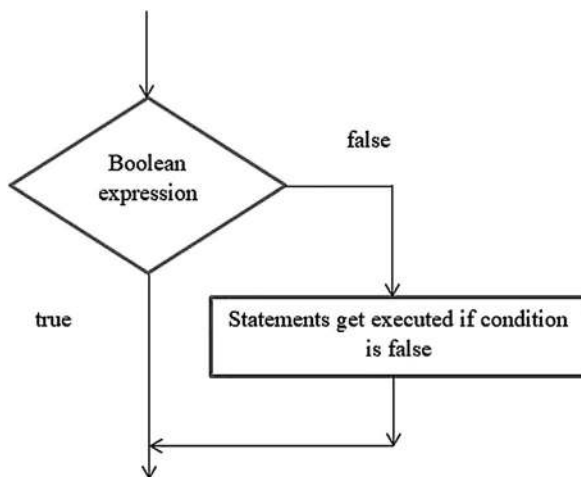
Syntax:

```

unless(boolean_expression)
{
    # will execute if a given condition is false
}

```

Flowchart:



Unless statement.

Example:

```
# program to illustrate
# unless statement

$c = 20;

unless($c != 20)
{

    # if condition is false then
    # print following
    printf "c is not equal to 20\n";

}
```

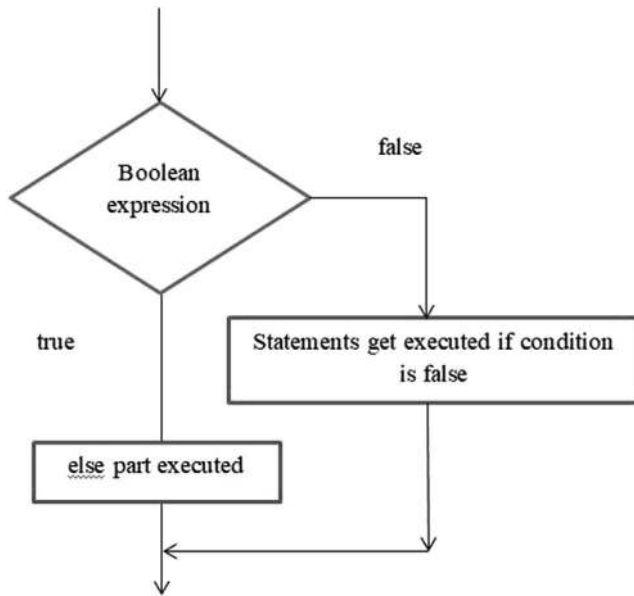
unless else Statement

When the boolean expression is true, the except statement is followed by an optional else statement.

Syntax:

```
unless(booleanexpression)
{
    # execute if a given condition is false
}

else
{
    # execute if a given condition is true
}
```

Flowchart:

Unless else Statement.

Example:

```

# program to illustrate
# unless - else statement

$c = 20;

unless($c == 20)
{
    # if condition is false then
    # print following
    printf "c is not equal to 20\n";
}

else
{

```

```

    # if condition is true then
    # print following
    printf "c is equal to 20\n";
}

```

unless elsif Statement

The unless and an optional elsif can follow statement...else statement, which allows us to test many conditions with a single unless...elsif statement.

Remember the following:

Unless statements can have any number of elsifs, and all of them must appear before the else.

Unless statements can have zero or one else's, and they must appear after any elsif statements.

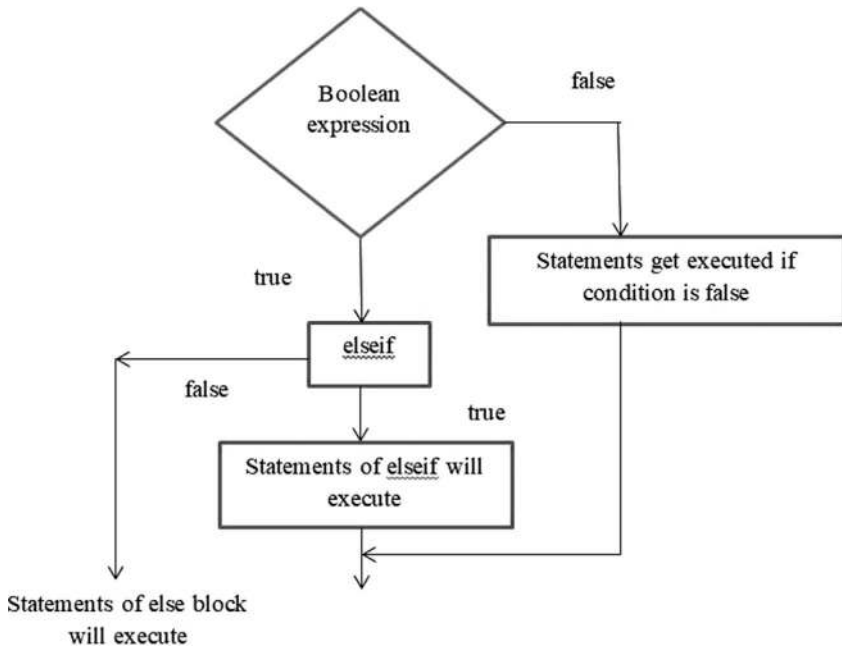
When an elsif succeeds, none of the other elsif's or elses are tried.

Syntax:

```

unless(boolean_expression 1)
{
    # Executes when boolean expression 1 is false
}
elsif( boolean_expression 2)
{
    # Executes when boolean expression 2 is true
}
else
{
    # Executes when none of above condition is
met
}

```

Flowchart:

Statement of unless elsif.

Example:

```

# program to illustrate
# unless elsif statement
$c = 70;

unless($c == 80)
{

# if condition is false
printf "c is not equal to 80\n";
}
elsif($c == 80)
{

# if condition is true
printf "c is equal to 80\n";
}
}

```

```

else
{

# if none of condition matches
printf "Value of c is %c\n";
}

```

LOOPS IN PERL

In programming languages, looping is a feature that allows the execution of a collection of instructions or functions repeatedly while some condition is true. Loops make the job of the coder easier. Perl supports many forms of loops to handle conditional situations in programs. Perl's loops are discussed below.

for Loop

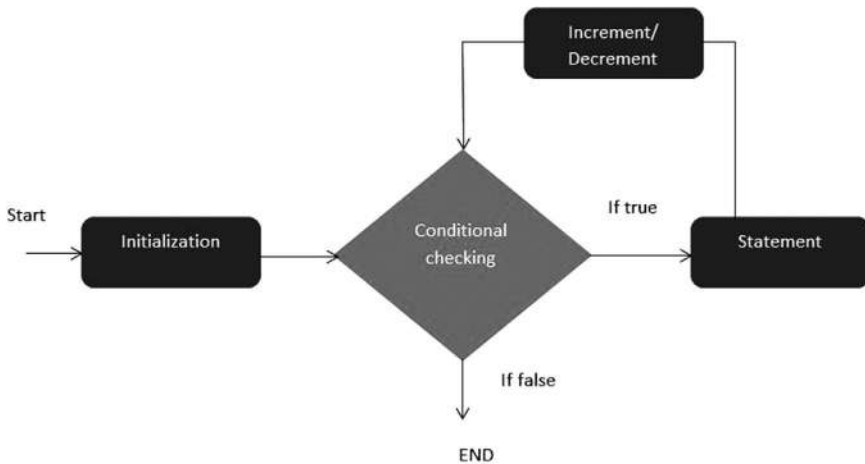
The “for” loop is a shorthand means of expressing the loop structure. Unlike a while loop, a for statement consumes the setup, condition, and increment/decrement on a single line, resulting in a shorter, easier to debug looping structure.

Syntax:

```

for (init statement; condition; increment/
decrement )
{
    # Code Executed
}

```

Flowchart:

for loop in Perl.

A for loop operates on a predefined control flow. The following factors influence control flow:

- **init statement:** The init statement is the initial statement that is performed. In this stage, we set up a variable to control the loop.
- **condition:** The specified condition is assessed in this stage, and the for loop is executed if it is True. Because the condition is verified before the loop statements are executed, it is also an entry control loop.
- **Statement execution:** When the condition is true, the statements in the loop body are performed.
- **increment/decrement:** The loop control variable is incremented or decremented here to update the variable for the next iteration.
- **Loop termination:** When the condition is met, the loop terminates, indicating the end of its life cycle.

Example:

```
# program to illustrate
# for loop
```

```
# for loop
for ($count = 1 ; $count <= 4 ; $count++)
{
    print "PeeksForPeeks\n"
}
```

foreach Loop

A foreach loop iterates over a list, and the variable stores the values of the list's items one at a time. It is mainly utilized when we have a collection of data in a list and wish to iterate through the list's elements rather than its range. The loop performs the iteration of each element automatically.

Syntax:

```
foreach variable
{
    # Code Executed
}
```

Example:

```
# program to illustrate
# foreach loop

# Array
@data = ('PEEKS', 'FOR', 'PEEKS');

# foreach loop
foreach $word (@data)
{
    print $word
}
```

while Loop

A while loop typically accepts an expression enclosed in parentheses. If the expression is True, the code within the while loop's body is performed. A while loop is used when we don't know how many times we want the loop to be performed but know the loop's termination condition. Because the condition is verified before performing the loop, it is also known as an entry controlled loop. The while loop is similar to a loop if statement.

Syntax:

```
while (condition)
{
    # Code executed
}
```

Example:

```
# program to illustrate
# while loop

# while loop
$count = 4;
while ($count >= 0)
{
    $count = $count - 1;
    print "PeeksForPeeks\n";
}
```

Infinite While Loop

A while loop can continue indefinitely, which means it has no end condition. In other words, some circumstances remain true, causing the while loop to continue endlessly or never finish.

Example: The following application will print the provided statement indefinitely and display the runtime error Output Limit Exceeded on the online IDE.

```
# program to illustrate
# infinite while loop

# infinite while loop
# containing condition 1
# which is always true
while(1)
{
    print "Infinite While Loop\n";
}
```

do...while loop

The only difference between a do...while loop and a while loop is that the do...while loop is executed at least once. After the initial execution, the

condition is verified. When we want the loop to execute at least once, we use a `do...while` loop. Because the condition is verified after performing the loop, it is also known as an exit controlled loop.

Syntax:

```
do {
    # statements Executed
} while(condition);
```

Example:

```
# program to illustrate
# do..while Loop

$c = 20;

# do..While loop
do {
    print "$c ";
    $c = $c - 1;
} while ($c > 0);
```

until Loop

The inverse of the while loop is the until loop. It accepts a condition in the parenthesis and only runs until it is false. It essentially repeats an instruction or sequence of instructions until the condition is FALSE. It is also an entry controller loop, which means that the condition is tested first, and then a sequence of instructions within a block is executed.

Syntax:

```
until (condition)
{
    # Statements executed
}
```

Example:

```
# program to illustrate until Loop

$c = 20;

# until loop
until ($c < 1)
{
    print "$c ";
    $c = $c - 1;
}
```

Nested Loops

A nested loop is a loop that is contained within another loop. Perl programming also supports nested loops. All of the loops listed above can also be nested.

Perl syntax for several nested loops:

- **Nested for loop**

```
for (init statement; condition; increment/
decrement )
{
    for (init statement; condition; increment/
decrement )
    {
        # Code Executed
    }
}
```

- **Nested foreach loop**

```
foreach variable_1 (@array_1) {

    foreach variable_2 (@array_2)
    {

        # Code Executed
    }
}
```

- **Nested while loop**

```
while (condition)
{
    while (condition)
    {
        # Code Executed
    }
}
```

- **Nested do...while loop**

```
do{
    do{

        # Code Executed

    }while(condition);

}while(condition);
```

- **Nested until loop**

```
until (condition) {

    until (condition)
    {
        # Code Executed
    }

}
```

Example:

```
# program to illustrate
# nested while Loop

$c = 5;
$d = 0;

# outer while loop
while ($c < 7)
{
    $d = 0;

    # inner while loop
    while ( $d <7 )
```

```

{
    print "value of c = $c, d = $d\n";
    $d = $d + 1;
}

$c = $c + 1;
print "Value of c = $c\n\n";
}

```

given-when STATEMENT

In Perl, a given-when statement replaces lengthy if statements that compare a variable to numerous integral values.

- The given-when statement is a branch statement with many options. It allows us to easily route execution to various areas of code based on the value of the expression.
- A control statement is provided that allows a value to alter execution control.

The switch-case in Perl is analogous to the switch-case in C/C++, Python, or PHP. It, too, substitutes numerous if statements with distinct cases, much like the switch statement.²

Syntax:

```

given(expression)
{
    when(value1) {Statement;}
    when(value2) {Statement;}

    default {# Code if no other case matchs}
}

```

given-when statements also include two extra keywords, break and continue. These keywords keep the program flowing and aid in exiting the program or skipping execution at a certain value.

break: The break keyword is used to exit a when block. There is no need to explicitly express the break after every when the block in Perl. It is already implicitly defined.

continue: On the other hand, if the first when the block is correct, continue proceed to the next when block.

A conditional statement in a given-when statement must not be repeated in multiple when statements since Perl only checks for the first occurrence of that condition and the subsequent repeating statements are disregarded. Furthermore, a default statement must be placed after all when statements since the compiler checks for condition matching with each when statement in order, and if we insert default in between, it will take a break and print the default statement.

Example:

```
#!/usr/bin/perl

# program to print respective day
# for daycode using given-when statement
use 5.010;

# Asking user to provide day-code
print "Enter daycode between 0-6\n";

# Removing newline using the chomp
chomp(my $daycode = <>);

# Using given-when statement
given ($daycode)
{
    when ('0') { print 'Sunday' ;}
    when ('1') { print 'Monday' ;}
    when ('2') { print 'Tuesday' ;}
    when ('3') { print 'Wednesday' ;}
    when ('4') { print 'Thursday' ;}
    when ('5') { print 'Friday' ;}
    when ('6') { print 'Saturday' ;}
    default { print 'Invalid daycode';}
}
```

Nested given-when Statement

given-when nested statement refers to given-when statements that are included within other given-when statements. This may be used to keep a hierarchy of user-supplied inputs for a given output set.

Syntax:

```

given(expression)
{
    when(value1) {Statement;}
    when(value2) {given(expression)
                  {
                      when(value3) {Statement;}
                      when(value4) {Statement;}
                      when(value5) {Statement;}
                      default{# Code if no other
case-matches}
                  }
                }
    when(value6) {Statement;}

    default {# Code if no other case-matches}
}

```

Here's an example of a Nested given-when statement:

```

#!/usr/bin/perl

# program to print respective day
# for the daycode using given-when statement
use 5.010;

# Asking user to provide daycode
print "Enter daycode between 0-6\n";

# Removing the newline using chomp
chomp(my $daycode = <>);

# Using given-when statement
given ($daycode)
{
    when ('0') { print 'Sunday' ;}
    when ('1') { print "What time of day is it?\n";
                chomp(my $daytime = <>);

                # Nested given-when statement
                given($daytime)
                {

```

```

                                when('Morning') {print 'It is
Monday Morning'};
                                when('Noon') {print 'It is Monday
noon'};
                                when('Evening') {print 'It is
Monday Evening'};
                                default{print'Invalid Input'};
                                }
                                }
    when ('2') { print 'Tuesday' ;}
    when ('3') { print 'Wednesday' ;}
    when ('4') { print 'Thursday' ;}
    when ('5') { print 'Friday' ;}
    when ('6') { print 'Saturday' ;}
    default { print 'Invalid daycode';}
}

```

When the Input day-code is anything other than 1, the code will not run the nested given-when block and the output will be the same as in the previous example; but, if we supply 1 as Input, the code will execute the nested given-when block and the result will differ from the previous example.

goto STATEMENT

The goto statement in the Perl is a jump statement, sometimes known as an unconditional jump statement. The goto statement can use to jump from one location within a function to another.

Syntax:

```

LABEL:
Statement-1;
Statement-2;
.
.
.
.
.
.
Statement-n;
goto LABEL;

```

The `goto` statement in the preceding syntax instructs the compiler to immediately go/jump to the statement designated as LABEL. The label is a user-defined identifier that specifies the target statement in this case. The destination statement is the statement that comes immediately after “label.”

In Perl, the `goto` statement can have three forms: Label, Expression, and Subroutine.

- **Label:** It will jump to the statement indicated by the LABEL and continue execution from there.
- **Expression:** There will be an expression in this form that will return a Label name after evaluation, and `goto` will cause it to jump to the labeled statement.
- **Subroutine:** `goto` will move the compiler from the currently executing subroutine to the subroutine of the provided name.

Syntax:

```
goto LABEL
```

```
goto EXPRESSION
```

```
goto Subroutine-Name
```

goto with LABEL name: LABEL name is used to jump to a specific statement in code and begin execution there. Its reach, however, is restricted. It can only function inside the scope of where it is invoked.

Example:

```
# Program to print numbers
# from 1 to 20 using goto statement

# function to print the numbers from 1 to 20
sub printNumbers()
{
    my $n = 1;
label:
    print "$n ";
    $n++;
}
```

```

        if ($n <= 20)
        {
            goto label;
        }
    }

# Driver-Code
printNumbers();

```

goto using Expression: An expression may also be used to call a specific label and transmit control to that label. When this expression is provided to the goto statement, it evaluates to a label name, and execution continues from the statement described by that label name.

Example:

```

# Program to print numbers
# from 1 to 20 using the goto statement

# Defining the two strings
# which contain
# label name in the parts
$c = "lab";
$d = "el";

# function to print numbers from 1 to 20
sub printNumbers()
{
    my $n = 1;
label:
    print "$n ";
    $n++;
    if ($n <= 20)
    {
        # Passing Expression to label name
        goto $c.$d;
    }
}

# Driver-Code
printNumbers();

```

goto with Subroutine: The goto command may also invoke a subroutine. Based on its usage, this procedure is called from within another subroutine or alone. It keeps the task to be done next to the calling statement. This method may recursively invoke a function to print a series or range of characters.

Example:

```
# Program to print numbers
# from 1 to 20 using goto statement

# function to print numbers from 1 to 20
sub label
{
    print "$n ";
    $n++;

    if($n <= 20)
    {
        goto &label;
    }
}

# Driver Code
my $n = 1;
label();
```

next OPERATOR

In Perl, the next operator skips the current loop execution and moves the iterator to the value specified by the next. If a label is specified in the program, execution proceeds to the next iteration designated by the Label.

Syntax:

```
next Label
```

First example:

```
#!/usr/bin/perl -w

# Program to find frequency
# of an element
```

```

@Array = ('P', 'E', 'E', 'K', 'S');
$d = 0;
foreach $key (@Array)
{
    if($key eq 'E')
    {
        $d = $d + 1;
    }
    next;
}

print "Frequency of E in Array: $d";

```

Second example:

```

#!/usr/bin/perl
$i = 0;

# label for the outer loop
outer:
while ( $m < 3 ) {

    $n = 0;
    while ( $n < 3 ) {

        # Printing values of i and j
        print "m = $m and n = $n\n";

        # Skipping loop if i==j
        if ( $n == $m ) {

            $m = $m + 1;
            print "As m == n, hence going back to
the outer loop\n\n";

            # Using next to skip an iteration
            next outer;
        }
        $n = $n + 1;
    }

    $m = $m + 1;

}# end of the outer loop

```

redo OPERATOR

In Perl, the redo operator begins from the specified label without evaluating the conditional expression. Once redo is called, no further statements in that block will execute. Even if there is a continue block, it will not be performed after the redo call. When a Label is used with the redo operator, the execution begins with the loop defined by the Label.

Syntax:

```
redo Label
```

Returns:

No Value

First example:

```
#!/usr/bin/perl -w

$x = 1;

# Assigning label to the loop
PFP: {
    $x = $x + 5;
    redo PFP if ($x < 10);
}

# Printing value
print ($x);
```

Second example:

```
#!/usr/bin/perl -w

$x = 1;

# Assigning label to the loop
$count = 1;
PFP: while($count < 10) {
    $x = $x + 5;
    $count++;
    redo PFP if ($x < 100);
}

# Printing value
print (" $x $count");
```

last IN LOOP

The last keyword is used in a loop control statement to make the current iteration of the loop the last. If a label is provided, it exits the loop through the label.

Syntax:

```
# Comes out of current loop.
last

# Comes out of loop specified by
# MY_LABEL
last MY_LABEL
```

First example:

```
#!/usr/bin/perl
$sum = 0;
$m = 0;
$n = 0;

while(1)
{

    $sum = $m + $n;
    $m = $m + 2;

    # Condition to end the loop
    if($sum > 20)
    {
        print "Sum = $sum\n";
        print "Exiting loop\n";
        last;
    }
    else
    {
        $n = $n - 1;
    }
}

print "Loop ended at Sum > 20\n";
```

Second example:

```
#!/usr/local/bin/perl

$m = 1;
$sum = 0;

# Outer Loop
Label1: while($m < 16)
{
    $n = 1;

    # Inner Loop
    Label2: while ($n < 8)
    {
        $sum = $sum + $n;
        if($m == 8)
        {
            print "Sum is $sum";

            # terminate the outer loop
            last Label1;
        }
        $n = $n * 2;
    }
    $m = $m * 2;
}
```

In this chapter, we covered control flow statements in Perl.

NOTES

-
1. Perl | Decision-Making (if, if else, Nested if, if elsif ladder, unless, unless else, unless elsif).
 2. Perl | given-when Statement.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

File Handling in Perl

IN THIS CHAPTER

- Introduction of File Handling
- Opening and Reading a File
- Writing and Appending to a File
- Reading a CSV File
- File Test Operators
- File Locking
- Use of Slurp Module
- Useful File-handling Functions

In the previous chapter, we discussed control flow statements, and in this chapter, we will cover file handling.

INTRODUCTION OF FILE HANDLING

A `FileHandle` in Perl connects a name with an external file that can use until the program terminates or the `FileHandle` is closed. In summary, a `FileHandle` is a connection that can be used to edit the contents of an external file, and the connection (the `FileHandle`) is given a name for easier access and convenience.

STDIN, STDOUT, and STDERR are the three fundamental FileHandles in Perl, representing standard input, standard output, and standard error devices, respectively.

The open function is typically used for file handling.

Syntax:

```
open(FileHandle, Mode, FileName);
```

Parameters:

FileHandle- The reference to file, that can use within the program or until its closure.

Mode- Mode in which file is to be opened.

FileName- The name of a file to be opened.

Mode and FileName can also combine to produce a single open expression.

Syntax:

```
open(FileHandle, Expression);
```

Parameters:

FileHandle- The reference to file, that can use within the program or until its closure.

Expression- The Mode and FileName clubbed together.

The close function is used to close FileHandle.

Syntax:

```
close(FileHandle);
```

Parameters:

FileHandle- The FileHandle to close.

Using FileHandle To Read and Write to a File

The print function can be used to read from a FileHandle.

Syntax:

```
print(<FileHandle>);
```

Parameters:

FileHandle- FileHandle opened in the read mode or similar mode.

The print function can also be used to write to a file.

Syntax:

```
print FileHandle String
```

Parameters:

FileHandle- FileHandle opened in the write mode or similar mode.

String- The String to insert in the file.

Various File Handling Modes

Mode	Explanation
"<"	The Read Only Mode
">"	Creates file (if necessary), Clears contents of the File and Writes to it
">>"	Creates file (if necessary), Appends to File
"+<"	Reads and Writes but does NOT Create
"+>"	Creates file (if necessary), the Clears, Reads and Writes
"+>>"	Creates file (if necessary), the Reads and Appends

Example: Consider the file Hello1.txt, which initially contains the text "Welcome to PeaksForPeeks!!".

1. Mode = "<"

This mode is read-only. This mode is used to read the file's content line by line.

```
#!/usr/bin/perl
```

```
# Opening a File in the Read-only mode
open(r, "<", "Hello1.txt");
```

```
# Printing content of File
```

```
print(<r>);
```

```
# Closing File
close(r);
```

2. **Mode = ">"**

This is the read-only mode. The original contents are erased when we open a file in this mode. If no file with the same name is found, it creates one.

```
#!/usr/bin/perl
```

```
# Opening File Hello1.txt in Read mode
open(r, "<", "Hello1.txt");
```

```
# Printing existing content of the file
print("Existing Content of Hello1.txt: ". <r>);
```

```
# Opening File in Write mode
open(w, ">", "Hello1.txt");
```

```
# Set r to the beginning of Hello1.txt
seek r, 0, 0;
```

```
print "\nWriting to the File...";
```

```
# Writing to Hello1.txt using print
print w "The Content of this file is changed";
```

```
# Closing FileHandle
close(w);
```

```
# Set r to beginning of Hello1.txt
seek r, 0, 0;
```

```
# Print the current contents of Hello1.txt
print("\nUpdated Content of Hello1.txt: ".<r>);
```

```
# Close FileHandle
close(r);
```

3. **Mode=">>"**

Append mode is active. The original content is not deleted when we open a file in this mode. Because the string always attaches at the

end, this mode cannot be used to overwrite. If no file with the same name is discovered, it creates one.

```
#!/usr/bin/perl

# Opening File Hello1.txt in Read mode
open(r, "<", "Hello1.txt");

# Printing existing content of the file
print("Existing Content of Hello1.txt: ". <r>);

# Opening the File in the Append mode
open(A, ">>", "Hello1.txt");

# Set r to beginning of Hello1.txt
seek r, 0, 0;

print "\nAppending to the File...";

# Appending to Hello1.txt using print
print A " Hello Everyone!";

# close FileHandle
close(A);

# Set r to the beginning of Hello1.txt
seek r, 0, 0;

# Print current contents of Hello1.txt
print("\nUpdated Content of Hello1.txt: ".<r>);

# Close FileHandle
close(r);
```

4. **Mode = "+<"**

This is known as Read-Write mode. This function may be used to replace an existing string in file. It is unable to create a new file.

```
#!/usr/bin/perl

# Open Hello1.txt in Read-Write Mode
open(rw, "+<", "Hello1.txt");

# Print original contents of File.
```

```

# rw is set to the end.
print("Existing Content of the Hello1.txt:
".<rw>);

# The string is attached at the end
# of original contents of the file.
print rw "Added using the Read -Write Mode.";

# Set rw to the beginning of File for reading.
seek rw, 0, 0;

# Printing Updated content of the File
print("\nUpdated contents of Hello1.txt: ".<rw>);

# Close FileHandle
close(rw);

```

5. **Mode = "+>"**

This is known as Read-Write mode. The distinction between "+" and "+>" is that "+>" can create a new file even if one with the same name already exists, whereas "+" cannot.

```

#!/usr/bin/perl

# Opening File Hello1.txt in the Read mode
open(r, "<", "Hello1.txt");

# Printing existing content of the file
print("Existing Content of Hello1.txt: ". <r>);

# Closing File
close(r);

# Open Hello1.txt in the Read-Write Mode
open(rw, "+>", "Hello1.txt");

# Original contents of File
# are cleared when File is opened
print("\nContents of Hello1.txt gets cleared..");

# The string is written to File
print rw "Hello!!! This is the updated file.";

```

```
# Set rw to the beginning of File for reading.
seek rw, 0, 0;

print("\nUpdated Content of Hello1.txt: " .<rw>);

# Closing File
close(rw);
```

6. Mode = "+>>"

This is called Read-Append mode. This may be used to both read from and append to a file. A new one with the same name is generated if no such file exists.

```
# Open Hello1.txt in Read-Append Mode
open(ra, "+>>", "Hello1.txt");

# Set ra to beginning of File for reading.
seek ra, 0, 0;

# Original content of File
# is NOT cleared when File is opened
print("Existing Content of File: ". <ra>);

print "\nAppending to File....";

# The string is appended to File
print ra "Added using Read-Append Mode";

# Set ra to the beginning of File for reading.
seek ra, 0, 0;

# Printing updated content
print("\nUpdated content of File: ". <ra>);

# Closing File
close(rw);
```

Redirecting Output

Using the `choose` function, output may be redirected away from the console and into a file.

Syntax:

```
select FileHandle;
Parameters:
```

FileHandle - FileHandle of File to be selected.

Steps:

- To write, open a FileHandle and type ">", ">>", "<", "<>", or "<>>".
- Using the select function, choose the FileHandle.

Anything printed using the print function is now redirected to the file.

Example:

```
# Open a FileHandle in Write Mode.
open(File, ">", "Hello1.txt");

# This sets File as default FileHandle
select File;

# Writes to File
print("This goes to File.");

# Writes to File
print File "\nThis goes to File too.";

# This sets STDOUT as default FileHandle
select STDOUT;
print("This goes to console.");

# Close FileHandle.
close(File);
```

FILE OPENING AND READING

A FileHandle is a Perl internal structure that connects a physical file to a name. All FileHandles have read/write access; therefore, reading/writing is possible once attached to a file. However, when associating a FileHandle, the mode in which the file handle is opened must give.

Opening a File

The `Open` function is used to open either a new or existing file.

Syntax:

```
open FILEHANDLE, VAR
```

In this case, `FileHandle` is the handle returned by the `open` function, and `VAR` is the expression containing the file name and opening mode.

The table below displays the various file opening modes and access to various operations.

Mode	Description
<code>r</code> or <code><</code>	The Read Only Access
<code>w</code> or <code>></code>	Creates, Writes, and Truncates
<code>a</code> or <code>>></code>	Writes, Appends, and Creates
<code>r+</code> or <code><+</code>	Reads and Writes
<code>w+</code> or <code>>+</code>	The Reads, Writes, Creates, and Truncates
<code>a+</code> or <code>>>+</code>	The Reads, Writes, Appends, and Creates

Reading a File

When a `FileHandle` is assigned to a file, it may perform actions such as reading, writing, and appending. There are several methods for reading the file.

- Making use of a `FileHandle` operator
- Utilizing the `getc` function
- Utilizing the `read` function

FileHandle Operator

The most common way to read data from an open `FileHandle` is with the operator `<>`. When used in a list context, the `<>` operator returns a list of lines from the provided `FileHandle`. The following example reads one line from a file and puts it in a scalar.

Let the contents of the file “PFP.txt” be as follows:

```
PeeksforPeeks
Hello Peek
Peek a revolution
Peeks are the best
```

Example:

```
# Opening file
open(fh, "PFP.txt") or die "File '$filename' can't
open";

# Reading First line from file
$firstline = <fh>;
print "$firstline\n";
```

getc Function

The `getc` function retrieves a single character from the FileHandle given, or STDIN if none is specified.

Syntax:

```
getc FILEHANDLE
```

Example:

```
# Opening file
open(fh, "PFP.txt") or die "File '$filename' can't
open";

# Reading First char from file
$firstchar = getc(fh);
print "$firstchar\n";
```

If an error occurs or the FileHandle is at the end of the file, it returns `undef`.

read Function

The `read` function reads binary data from a file using the FileHandle.

Syntax:

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

If no `OFFSET` is supplied, `LENGTH` denotes the length of data to be read, and the data is inserted at the beginning of `SCALAR`. Otherwise, data is inserted after `OFFSET` bytes in `SCALAR`. If the file reading succeeds,

the function returns the number of bytes read, zero at the end of the file, or undef if an error occurred.

Reading More than One Line at a Time

The following example reads the file's contents indicated by FileHandle until it hits the end of file (EOF).

Example:

```
# Opening the file
open(FH, "PFP.txt") or die "Sorry, couldn't open";
print "Reading the file \n";

# Reading file till FH reaches EOF
while(<FH>)
{
    # Printing one line at a time
    print $_;
}
close;
```

Exception Handling in Files

The exception can be handled in one of the two ways:

- If the file cannot open, throw an exception.
- If the file cannot open, issue a warning and continue operating.

Throw an Exception

When FileHandle is not allocated a valid file pointer, a die is run, which prints the message and terminates the current program.

Example:

```
# Initializing filename
$filename = 'PFP1.txt';

# Prints an error and exits if the file not found
open(fh, '<', $filename) or die "Couldn't Open the
file $filename";
```

The code produces an error and quits if the file is not found.

Give a Warning

When FileHandle cannot be assigned a proper file reference, it produces a warning message and continues to operate.

Example:

```
# Initializing filename
$filename = 'PFP.txt';
# Opening file and reading content
if(open(fh, '<', $filename))
{
    while(<fh>)
    {
        print $_;
    }
}

# Executes if the file is not found
else
{
    warn "Couldn't Open file $filename";
}
```

WRITING TO A FILE

A FileHandle is a variable used to read and write files. This FileHandle is linked with the file.

To write to a file, open it in write mode, as shown below:

```
open (FH, '>', "file_name.txt");
```

If the file already exists, it replaces the old content with the new content. Otherwise, a new file will generate with fresh content.

print() Function

The print() method is used to write data to a file.

Syntax:

```
print the filehandle string
```

When the file is opened, the FileHandle is associated with it, and the string contains the content to be written to the file.

Example:

```
# Opening file Hello1.txt in write mode
open (fh, ">", "Hello1.txt");

# Getting the string to be written
# to the file from the user
print "Enter the content to add\n";
$a = <>;

# Writing to file
print fh $a;

# Closing the file
close(fh) or "Couldn't close file";
```

The program operates as follows:

- Step 1: Open the Hello.txt file in write mode.
- Step 2: Extract text from the standard input keyboard.
- Step 3: Save the string saved in “\$a” to the file indicated by the FileHandle “fh”.
- Step 4: Save the file.

The following example reads the source file’s content and writes it to the destination file:

```
# Source-File
$src = 'Source.txt';

# Destination File
$des = 'Destination.txt';

# open the source file for reading
open(FHR, '<', $src);

# open the destination file for writing
open(FHW, '>', $des);

print("Copying the content from $src to $des\n");
```

```

while(<FHR>)
{
print FHW $_;
}

# Closing filehandles
close(FHR);
close(FHW);

print "File content copied successfully!\n";

```

The program operates as follows:

- Step 1: Read the file Source.txt and write the file Destination.txt.
- Step 2: Reading content from the FHR, which is a FileHandle for reading content and a FileHandle for writing content to a file.
- Step 3: Using the print function, copy the content.
- Step 4: Once file has been read, close the conn.

Error Handling and Error Reporting

Errors can be handled in one of the two ways:

- If the file cannot open, throw an exception (handling an error).
- If the file cannot open, issue a warning and continue operating (error reporting).

Throw an Exception (Using Die Function)

When FileHandle is not allocated a valid file pointer, a die is run, which prints the message and terminates the current program.

Example:

```

# Initializing filename
$filename = 'Hello1.txt';
# $filename = 'ello1.txt';

# Prints error and exits
# if file not found
open(fh, '<', $filename) or
    die "Couldn't Open the file $filename";

```

When a file exists in the given code, it is performed with no issues, but if the file doesn't exist, an error is generated, and the code ends.

Give a Warning (Using Warn Function)

When FileHandle cannot be assigned a proper file reference, it produces a warning message and continues to operate.

Example:

```
# Initializing filename
$filename = 'PFP.txt';

# Opening file and reading content
if(open(fh, '<', $filename))
{
    while(<fh>)
    {
        print $_;
    }
}

# Executes if the file is not found
else
{
    warn "Couldn't Open file $filename";
}
```

APPENDING TO A FILE

When a file is opened in write mode with ">," the existing file's content is destroyed, and content added with the print statement is written to the file. In this mode, the writing point is set to the end of the file. So the old file content is preserved, and everything written to the file using the print statement is appended to the end of the file. However, a read operation cannot execute until the file is opened in +>> mode, indicating add and read.

Example:

```
# Opening a file in the read mode
# to display existing content
open(FH, "Hello1.txt") or
```

```

        die "Sorry, couldn't open";

# Reading and printing existing
# content of the file
print "\nExisting Content of the File:\n";
while(<FH>)
{
    print $_;
}

# Opening file in append mode
# using >>
open(FH, ">>", "Hello.txt") or
die "File couldn't be opened";

# Getting the text to be appended
# from the user
print "\n\nEnter the text to append\n";
$a = <>;

# Appending content to file
print FH $a;

# Printing the success message
print "\nAppending to File is Successful.\n";

# Reading the file after appending
print "\nAfter the appending, Updated File is\n";

# Opening file in the read mode to
# display updated content
open(FH, "Hello1.txt") or
    die "Sorry, couldn't open";
while(<FH>)
{
    print $_;
}
close FH or "couldn't close";

```

The program operates as follows:

- Step 1: Open a file in read mode to view the file's existing content.
- Step 2: Print the file's existing content.

- Step 3: Add material to the file by opening it in Append mode.
- Step 4: Collect user text to attach to a file.
- Step 5: Add text to the file.
- Step 6: Re-read the file to determine if the content has been changed.
- Step 7: File closure.

CSV FILE READING

Perl was originally intended for text processing, such as extracting information from a text file and converting it to a new format. In Perl, reading a text file is a pretty frequent activity. For example, reading CSV (comma-separated value) files to extract data and information is a common task.

A CSV file can be created using any text editor, such as notepad, notepad++, or others. After adding material to a notepad text file, save it as a CSV file with the .csv extension.

A CSV file example:

	A	B	C	D
1	Danish	M	18	
2	Jadeep	M	20	
3	Emma	F	21	
4				
5				

CSV file example.

A CSV file can be used to handle database record files for an enterprise or organization. These files can be opened with ease in Excel and can be edited with any compatible application. Perl also enables the processing and construction of these “csv” files by extracting values from the file, modifying these values, and inserting the modified values back into the file. We will use the split function to extract every value from a particular line.

Use of Split() for Data Extraction

split() is a standard Perl function used to divide a string into sections using a delimiter. This delimiter can be any character specified by the user; however, we usually use a comma.

split() accepts two arguments. The first is a delimiter, and the second is the string to be divided.

Syntax:

```
split(_delimiter_, _string_);
```

Parameter:

`_delimiter_` : Separator value between the elements

`_string_` : From which values are to extract

Returns: Array of string elements separated by

`_delimiter_`

Example:

Input: `$s = "Shona loves Sugar"`

Output: `"Shona", "loves", "Sugar"`

If Input string is passed to `split` function as,

```
@words = split("", $s);
```

The array `@words` will fill with 3 values: `"Shona", "loves" and "Sugar"`.

The following procedures are used to divide lines in a CSV file into sections using a delimiter:

- Step 1: Read file line by line.
- Step 2: Put all the values from each line into an array.
- Step 3: To obtain the result, print out all the data individually.

Let's look at an example to grasp the subject better. The `split()` method is used to divide the strings recorded in the `new.csv` file using a delimiter, as seen below:

```
use strict;
```

```
my $file = $ARGV[0] or die;
```

```
open(my $data, '<', $file) or die;
```

```
while (my $line = <$data>)
```

```
{
```

```
    chomp $line;
```

```
    # Split line and store it
```

```
    # inside words array
```

```
    my @words = split " ", $line;
```

```

for (my $x = 0; $x <= 2; $x++)
{
    print "$words[$x] ";
}
print "\n";
}

```

Save the preceding code in a text file with the extension .pl. We'll store it as test1.pl in this case.

Use the following command to run the previously stored file:

```
perl test1.pl new.csv
```

Character Escaping a Comma

Sometimes a file has a comma inside the fields of a string that, if deleted, changes the interpretation of the data or renders the record unusable. If a split() function is used in this case, even if it is within quotes, it will divide the data each time it encounters a comma as a delimiter because split() does not care about quotes and does not understand CSV. It just cuts when it encounters the separator character.

The following is a CSV file with a comma between the quotes:

	A	B	C	D
1	Danish	M	18	
2	Jaydeep	M	20	
3	Emma	F	21	
4	Mr, Daman	M	24	
5				

Character escaping a comma.

The first field in the above CSV file includes a comma; thus, it is closed within quotes. However, if we execute the split() method on this file, it will ignore any such quotes.

The split() method in the above program broke the string field into pieces even though it was between quotes; moreover, because we were printing just three fields in our code, the third field of the last string was discarded in the output file.

To handle such scenarios, Perl has several constraints and scopes that allow the compiler to bypass the division of fields within quotes.

We utilize TEXT::CSV, which provides a full CSV reader and writer. TEXT::CSV is a Perl MCPAN module that adds numerous additional features such as reading, parsing, and writing CSV files. The following pragma can be used to include these modules in the Perl program:

```
use Text::CSV
```

However, we must first download and install this module in our device to use its features.

Installation of the TEXT::CSV

For Windows:

```
perl -MCPAN -e shell
install Text::CSV
```

For Debian/Ubuntu-based system:

```
$ sudo apt-get install libtext-csv-perl
```

For RedHat/Centos/Fedora-based system:

```
$ sudo yum install perl-Text-CSV
```

To escape the comma character between quotes, run the following code on our new.csv file:

```
use strict;

# Using the Text::CSV file to allow
# full CSV Reader and Writer
use Text::CSV;

my $csv = Text::CSV->new({ sep_char => ',', ' ' });

my $file_to_be_read = $ARGV[0] or die;

# Reading file
open(my $data_file, '<', $file_to_be_read) or die;
while (my $line = <$data_file>)
```

```

{
chomp $line;

# Parsing line
if ($csv->parse($line))
{

    # Extracting elements
    my @words = $csv->fields();
    for (my $x = 0; $x <= 2; $x++)
    {
        print "$words[$x] ";
    }

    print "\n";
}
else
{
    # Warning to display
    warn "Line could not parse: $line\n";
}
}

```

The first field in the above example now includes a comma that was escaped while parsing the CSV file.

```
my $csv = Text::CSV->new({ sep_char => ',', ' ' });
```

The preceding line describes how to invoke the class's constructor. The arrow `->` is used to invoke a constructor.

```
$csv->parse($line)
```

This call will attempt to parse the current line and divide it into chunks. Depending on success or failure, return true or false.

Fields with Newlines Embedded

Specific fields in a CSV file may be multi-lined or have a newline embedded between the words. When these multi-lined fields are given via a `split()` method, they behave considerably differently from other files with no embedded newline.

Example:

	A	B	C
1	Danish	M	18
2	Jaydeep	M	20
3	Emma	F	21
4	Mr,	M	24

Fields with newlines embedded.

To handle such files, Perl provides the `getline()` function.

```
use strict;

# Using the Text::CSV file to allow
# full CSV Reader and Writer
use Text::CSV;

my $file = $ARGV[0] or die;

my $csv = Text::CSV->new (
{
    binary => 1,
    auto_diag => 1,
    sep_char => ',',
});

my $sum = 0;

# Reading file
open(my $data, '<:encoding(utf8)', $file) or die;

while (my $words = $csv->getline($data))
{
    for (my $x = 0; $x < 3; $x++)
    {
        print "$words->[$x]";
    }
    print "\n";
}

# Checking for the End-of-file
if (not $csv->eof)
```

```

{
    $csv->error_diag();
}
close $data;

```

The embedded newline in the above CSV file is now handled with the `getline()` function, and Perl interprets the new field as one, as needed by the coder, and so was enclosed in quotes.

FILE TEST OPERATORS

In Perl, file test operators are logical operators that produce True or False values. Perl provides a plethora of operators for testing various features of a file. For example, the `-e` operator is used to check for the existence of a file. Alternatively, it can determine whether a file can write to before conducting the add operation. This will aid in lowering the number of mistakes a program may encounter.

The following example uses the “`-e`”, existence operator, to determine whether or not a file exists:

```

#!/usr/bin/perl

# Using the predefined modules
use warnings;
use strict;

# Providing path of the file to a variable
my $filename = 'C:\Users\PeeksForPeeks\PEP.txt';

# Checking for file existence
if(-e $filename)
{
    # If the File exists
    print("File $filename exists\n");
}

else
{
    # If File doesn't exists
    print("File $filename doesn't exists\n");
}

```

This file test operator `-e` takes a filename or FileHandle as an input. The following table lists the most important file test operators:

Operator	Description
<code>-r</code>	checks if file is readable
<code>-w</code>	checks if file is writable
<code>-x</code>	checks if file is executable
<code>-o</code>	checks if file is owned by effective uid
<code>-R</code>	checks if the file is readable by real uid
<code>-W</code>	checks if the file is writable by real uid
<code>-X</code>	checks if the file is executable by real uid/gid
<code>-O</code>	checks if file is owned by real uid
<code>-e</code>	checks if file exists
<code>-z</code>	checks if file is empty
<code>-s</code>	checks if file has nonzero size (returns size in bytes)
<code>-f</code>	checks if the file is a plain text file
<code>-d</code>	checks if the file is a directory
<code>-l</code>	checks if file is a symbolic link
<code>-p</code>	checks if file is a named pipe (FIFO); or FileHandle is a pipe
<code>-S</code>	checks if file is a socket
<code>-b</code>	checks if a file is a block special file
<code>-c</code>	checks if a file is a character special file
<code>-t</code>	checks if file handle is opened to a tty
<code>-u</code>	checks if file has setuid bit set
<code>-g</code>	checks if file has setgid bit set
<code>-k</code>	checks if file has sticky bit set
<code>-T</code>	checks if file is an ASCII text file (heuristic guess)
<code>-B</code>	checks if file is a “binary” file (opposite of <code>-T</code>)

We may use the AND logical operator in combination with the following file test operators:

```
#!/usr/bin/perl

# Using the predefined modules
use warnings;
use strict;

# Providing path of the file to a variable
my $filename = 'C:\Users\PeeksForPeeks\PEP.txt';

# Applying the multiple Test Operators
```

```
# on the File
if(-e $filename && -f _ && -r _ )
{
print("File $filename exists and readable\n");
}

else
{
    print("File $filename does not exists")
}
```

The above example checks for the existence of the file, as well as whether it is plain or not and if it is readable.

FILE LOCKING

File locking, or locking in general, is simply one of the several techniques offered to address resource sharing issues.

Locking the security and integrity of any document is a means of protecting it. The basic goal of file locking is to allow users to make changes to a document without causing havoc. When two or more people attempt to change the same file, problems may occur.

Consider the following example: a file holding project data. The file can be modified by the whole project team. On the web, a CGI script will be written to accomplish the following:

```
$file = "project.docx";
$commit = $ENV{'QUERY_INFO'};
open(FILE, "$file"); #opening document
while() {
if (m/^^$commit$/) {
    print "The Change already made\n";
    exit;
}
}
close(FILE);
push(@newcommit, $commit);
open(FILE, ">$file");
print ...
close(FILE);
print "Commit-made";
exit;
```

The locking of a file is performed at the system level; thus, the user need not be concerned with the specifics of applying the lock. The purpose of file locks is to put temporary limitations on certain files to limit how they may share between programs. Two sorts of locks are developed based on the nature of a given operation.

A shared lock is the first kind, whereas an exclusive lock is the second. Multiple processes may share read access to a file since read access does not alter the shared resource's state. Therefore, keeping a consistent picture of the shared resource is possible. The flock command may use in Perl to lock files.

flock()

flock() has two arguments. The FileHandle is the first. The second input specifies the locking procedure that is required.

Syntax:

```
flock [FILEHANDLE], [OPERATION]
```

OPERATION is a number value that might be 1, 2, 4, or 8.

These numerical numbers have many meanings and are used to execute various operations such as:

```
LOCK_SH
LOCK_EX
LOCK_NB
LOCK_UN
```

Perl employs numbers to represent values:

```
sub LOCK_SH { 1 } ## set as the shared lock
sub LOCK_EX { 2 } ## set as the exclusive lock
sub LOCK_NB { 4 } ## set the lock without any blocks
sub LOCK_UN { 8 } ## unlock FILEHANDLE
```

- **Shared lock:** A shared lock can be used when we just want to read a file while allowing others to view it. Shared lock not only creates a lock, but it also checks for the presence of additional locks. This lock does not cover the existence check for all locks, just the “Exclusive Lock.” If an exclusive lock is present, it will wait until the lock is

removed. It will be performed as shared lock after removal. Multiple shared locks may exist at the same time.

Syntax:

```
flock(FILE, 1); #from above code
```

- **Exclusive lock:** When a file is to be utilized by a group of individuals and everyone has the ability to make changes, an exclusive lock is employed. Only one exclusive lock will be placed on a file, allowing only one user/process to make modifications at a time. The rule of an exclusive lock is “I am the only one.” flock() searches the script for any additional sorts of locks. If it is discovered, it will remain until all of them have been deleted from the script. It will lock the file at the appropriate time.

Syntax:

```
flock(FILE, 2); #from above code
```

- **Non-blocking lock:** A non-blocking lock alerts the system that it does not need to wait for other locks to be released from the file. If another lock is detected in the file, it will produce an error message.

Syntax:

```
flock(FILE, 4); #from above code
```

- **Unblocking:** The same as the close(FILE) function, this function unblocks any given file.

Syntax:

```
flock(FILE, 8); #from above code
```

The following problem demonstrates the use of the flock() function:

PROBLEM 5.1

The main issue with the script described in the preceding example is when it opens the file – project.docx, clearing the first file and making it empty.

Consider Persons A, B, and C attempting to commit almost simultaneously.

Person A examines the data in the file. Closes it and begins editing the data (finishing the first file).

Meanwhile, Person B opens the file to inspect/read it and discovers it is empty.

That is a collision! The entire system will disrupt.

SOLUTION

This is where file locking is implemented. Person A, Person B, and Person C are willing to commit to the document.

Person A opens the file, examines its data, and then closes it (shared lock is enabled) (unlocking the file). Person A then wants to commit changes to the file and re-opens the file to make adjustments (this is when the exclusive lock comes into action). While person A is making changes to the file, neither person B nor person C can read or write from the file.

Person A completes his work and closes the file (unlocking it).

Person B would have received the error “File being viewed by another candidate” if they attempted to open the file in the interim. Consequently, there is no treading on one another’s toes, and the work flows smoothly.

flock() vs lockf()

In contrast to flock(), which locks entire files at once, the lockf() method locks fragments of a file. lockf() cannot be used directly in Perl, but Perl natively supports the flock system function but does not apply to network locks.

Perl supports the fcntl() system function, providing the most excellent locking controls.

SLURP MODULE

The File::Slurp module reads a file’s contents and saves them in a string. It is a simple and quick method of reading/writing/modifying entire files. It, like its name suggests, lets us to read or write whole files with a single call.

By importing this module into our program, the user may use functions like read_file, read_text, write_file, and so on to read and write data to and from files.

Installation of the File::Slurp module: We must first include this module in our Perl language package to utilize it. This can be accomplished by entering the following instructions into our Perl Terminal and installing the necessary module.

Step 1: Run the following command in our terminal:

```
perl -MCPAN -e shell
```

After entering the cpan shell, proceed to the following step to install the File::Slurp module.

Step 2: To install the module, run the following command:

```
install File::Slurp
```

The File::Slurp module will be installed as a result of this.

Step 3: To exit the cpan> prompt, type and execute the “q” command.

read_file function in the Slurp: The read_file function of File::Slurp reads the complete contents of a file with the file name and returns it as a string. However, File::Slurp is recommended since it has a few encoding layer flaws that may cause compilation difficulties. File::Slurper seeks to offer a viable answer to the difficulties above.

Syntax:

```
use File::Slurp;
my $text = read_file($file_name);
Return: It returns a string.
```

read_text function in the Slurp: The read_text method in File::Slurper accepts an optional encoding parameter (if any) and can automatically decode CRLF line ends if we request it (for Windows files).

Syntax:

```
use File::Slurper;
my $content = read_text($file_name);
Return: It returns string.
```

Note: CRLF line endings indicate a line break in a text file (Windows line break types).

Slurp’s write_file function: The File::Slurp module’s write_file method is used to write to many files at once. It writes to file using a scalar variable containing the content of another file read by the read_file function.

Syntax:

```
use File::Slurp;
write_file($file_name, $content);
```

Returns: It does not return any value, just writes content to the file.

First example: Storing file content using scalar

```
# Perl code to illustrate slurp function
use File::Slurp;

# read whole file into a scalar
my $content = read_file('C:\Users\PeeksForPeeks\PFP_Slurp.txt');

# write out a whole file from scalar
write_file('C:\Users\PeeksForPeeks\Copyof_PFP_Slurp.txt', $content);
```

Explanation: In the preceding Perl code, we used a slurp function to read a file named PFP_Slurp.txt containing several lines of text as input into a scalar variable named \$content and then wrote the contents of the file as a single string into another file Copyof_PFP_Slurp.txt.

Second example: Storing file content in an array

```
# perl code to illustrate slurp function
use File::Slurp;

# read whole file into a scalar
my @lines = read_file('C:\Users\PeeksForPeeks\PFP_Slurp2.txt');

# write out a whole file from scalar
write_file('C:\Users\PeeksForPeeks\Copyof_PFP_Slurp2.txt', @lines);
```

Explanation: In the preceding Perl code, we used a slurp function to read a file named PFP_Slurp2.txt containing an array of lines of text as input into an array variable named @lines and then wrote the contents of the whole file as a single string into a file named Copyof PFP_Slurp2.txt.

Third example: Writing a function that use the slurp technique

```
# Perl code to illustrate the slurp function
use strict;
use warnings;
use File::Slurp;

# calling user defined function
get_a_string();

sub get_a_string
{
# read entire file into a scalar
my $pfp_str = read_file('C:\Users\PeeksForPeeks\PFP_
User_Slurp.txt');

# write entire file from scalar
write_file('C:\Users\PeeksForPeeks\Copyof_PFP_User_
Slurp.txt', $pfp_str);
}
```

Explanation: In the above Perl code, strict and warnings allow the user to enter code more freely and catch errors such as typos in variable names earlier. We invoked a user-defined function named `get_a_string`, which then performs the slurp function, which reads a file containing some lines of text as input into a variable named `pfp_str` and then wrote the contents of the entire file as a single string into a file.

USEFUL FILE-HANDLING FUNCTIONS

Perl was initially developed for text processing, such as extracting information from a text file and converting the text file into a new format. These procedures can be carried out using a variety of built-in file functions.

Example:

```
#!/usr/bin/perl

# Opening File in Read-only mode
open(fh, "<", "File_to_be_read.txt");

# Reading next character from file
$ch =getc(fh)
```

```
# Printing read character
print "Character read from file is $ch";

# Closing File
close(fh);
```

The following are some helpful Perl routines for working with files:

Function	Description
glob()	Used to print the files in a directory passed as an input.
tell()	Obtains the location of the read pointer in a file using the FileHandle.
getc()	Used to read the next character from the file specified by the FileHandle argument.
reverse()	When used in list context, returns the list in reverse order; when used in scalar context, returns a concatenated string of the list's values, with each character in the string in the opposite order.
rename()	Renames a file from its previous name to a new one specified by the user.

In this chapter, we covered file handling in Perl with its syntax and relevant examples.

Regular Expressions in Perl

IN THIS CHAPTER

- Regular Expressions
- Operators
- Regex Character and Special Character Classes
- Quantifiers and Backtracking
- “e” Modifier in Regex
- “ee” Modifier
- pos() Function

In the previous chapter, we discussed file handling; in this chapter, we will cover regular expressions (Regex).

In Perl, a regular expression (Regex, Regexp, or RE) is a particular text string that describes a search pattern inside a given text. Regex in Perl is tied to the host language and is not the same as in PHP, Python, and other programming languages.

It is also known as “Perl 5 Compatible Regular Expressions” sometimes. Binding operators such as “=~” (Regex Operator) and “!~” (Negated Regex Operator) are used to utilize the Regex. Let’s start with creating patterns before moving on to binding operators.

Pattern construction: In Perl, patterns may be built using the `m//` operator. This operator simply inserts the required pattern between the two slashes, and the binding operators are used to search for the pattern in the provided text.

Using `m//` with binding operators: Binding operators are commonly used with the `m//` operator to match the desired pattern. To match string using a regular expression, use the `regex` operator. The left-hand side of the statement will include a string that will match the provided pattern on the right-hand side. The negated `regex` operator determines whether the string is equal to the regular expression supplied on the right-hand side.

Program 1: The following code demonstrates the use of “`m//`” and “`=~`”:

```
# program to demonstrate
# m// and =~ operators

# Actual String
$a = "PEEKSFORPEEKS";

# Prints match found if
# its found in $a
if ($a =~ m[PEEKS])
{
    print "Match is Found\n";
}

# Prints match not found
# if it is not found in $a
else
{
    print "Match is Not Found\n";
}
```

Program 2: To demonstrate the usage “`m//`” and “`!~`,” consider the following:

```
# program to demonstrate
# m// and !~ operators

# Actual String
$a = "PEEKSFORPEEKS";
```

```
# Prints match found if
# it is not found in $a
if ($a !~ m[PEEKs])
{
    print "Match is Found\n";
}

# Prints match not found
# if it is found in $a
else
{
    print "Match is Not Found\n";
}
```

Regular expression uses:

- It can be used to count the number of times a particular pattern appears in a string.
- It can be used to find a string that matches a particular pattern.
- It can also replace searched pattern with another string.

OPERATORS IN REGULAR EXPRESSION

The regular expression is a string that is a mixture of distinct characters that facilitates text string matching. A regex or regexp is another name for a regular expression.

The most basic way to utilize a regular expression is to use the binding operators `= ~` (Regex Operator) and `!~` (Negated Regex Operator).

In Perl, there are three types of regular expression operators:

- Match regular expression
- Substitute regular expression
- Global character transliteration regular expression

1. Pattern matching or regular expression matching: The match operator `m//` compares a string or statement to a regular expression. The forward slash in the operator (`m//`) serves as the delimiter, and this delimiter can also be `m{}`, `m()`, `m><`, and so on. The expression is written between the operator's two forward slashes.

Syntax: `m/PATTERN/`

PATTERN is the regular expression that will search in the string.

Let's look at some examples of pattern matching.

In the following examples, a string and a regular expression are matched, and if successful, "match found" is returned; otherwise, "match not found" is returned.

First example:

```
#!/usr/bin/perl

# Initializing string
$a = "PeeksforPeeks";

# matching the string and
# a regular expression and returns
# match found or not
if ($a =~ m/for/)
{
    print "Match is Found\n";
}
else
{
    print "Match is Not Found\n";
}
```

Second example:

```
#!/usr/bin/perl

# Initialising string
$a = "PeeksforPeeks";

# matching string and
# a regular expression and returns
# match is found or not
if ($a =~ m:abc:)
{
    print "Match is Found\n";
}
else
{
    print "Match is Not Found\n";
}
```

In the preceding code, an alternative delimiter, “:,” is used instead of “/,” demonstrating that the usage of “/” as a delimiter is not required.

2. Make a substitution (search and replace) regular expression: The substitute operator “s///” is used to search for and replace a specific word with a given regular expression. The delimiter is the forward slash in the operator (s///).

Syntax: s/PATTERN/REPLACEMENT/;

PATTERN is the regular expression that the REPLACEMENT regular expression will substitute.

Let’s look at a few instances of replacement Regex:

In the following cases, a PATTERN word is searched first, followed by a REPLACEMENT word.

First example:

```
#!/user/bin/perl

# Initialising a string
$string = "PeeksforPeeks is a computer science
portal.";

# Calling substitute regular expression
$string =~ s/PeeksforPeeks/pfp/;
$string =~ s/computer science/cs/;

# Printing substituted string
print "$string\n";
```

Second example:

```
#!/user/bin/perl

# Initialising string
$string = "10001";

# Calling substitution regular expression
$string =~ s/000/999/;

# Printing substituted string
print "$string\n";
```

3. Global character regular expression for transliteration: To replace all instances of a character with a single character, use the translation or transliteration operator “tr///” or “y///.” The delimiter is the forward slash used in the operator (tr/// and y///).

Syntax:

```
tr/SEARCHLIST/REPLACEMENTLIST/  
y/SEARCHLIST/REPLACEMENTLIST/
```

SEARCHLIST is the character, the occurrences of which will be replaced with the character in REPLACEMENTLIST.

Let’s look at a few instances of translation of Regex.

All instances of “G” in the following examples are replaced with “g” using two separate operators “tr///” and “y///”.

First example:

```
#!/user/bin/perl  
  
# Initialising string  
$string = 'PeeksforPeeks';  
  
# Calling tr/// operator  
$string =~ tr/P/p/;  
  
# Printing replaced string  
print "$string\n";
```

Second example:

```
#!/user/bin/perl  
  
# Initialising string  
$string = 'PeeksforPeeks';  
  
# Calling y/// operator  
$string =~ y/P/p/;  
  
# Printing replaced string  
print "$string\n";
```

REGEX CHARACTER CLASSES

To match the string of characters, character classes are employed. These classes allow the user to match any range of characters that the user does not know ahead of time. The set of characters to be matched is always enclosed by square brackets []. A character class will always be a perfect match for one character. If no match is detected, the entire regex matching fails.

Example:

If we have a number of strings like `#p#`, `#e#`, `#k#`, `#k#`, `#s#`, `#.#`, or `#@#` and we need to match a `#` character followed by “p,” “e,” “k,” “s,” “.” or “@,” then try the regex `/[#peeks@.#]/` that will match the required. It will begin a match with `#`, then match any character in `[]`, followed by another `#`. This regex will not match `##` or `#pe#` or `#pp#` or similar characters because, as previously stated, the character class always matches precisely one character between the two “#” characters.

Important notes:

- Inside the character class, the `Dot(.)` has lost its particular meaning, which was “everything except a newline.”
- Only inside a character class, the `Dot(.)` may match a single dot(.).
- The majority of special characters lose their particular significance within a character class; however, certain characters gain some special meaning within a character class.

Example:

```
# program to demonstrate
# character class

# Actual-String
$str = "#g#";

# Prints match found if
# it is found in $str
if ($str =~ /[#peeks@.#]/)
{
    print "Match is Found\n";
}
```

```
# Prints match is not found
# if it is not found in $str
else
{
    print "Match is Not Found\n";
}
```

Range in character class: Matching a long list of characters is tough to enter since the user may skip one or two characters. So, to make things easier, we'll utilize range. A dash(-) is commonly used to denote the range.

Example:

To specify range [abcdef] we can use /[a-f]/

Important notes:

- The -(dash) sign is used to specify a range.
- The user may also mix various ranges of characters, digits, and so on, such as [0-9a-gA-g]. Here, “-” allows the user to select any number of characters or digits from the range.
- If a user wants to match a dash (-) in a string, he may simply insert it between square brackets [].
- To match a closing square bracket in a string, just precede it with \ i.e. \] and place it between the square brackets [].

Example:

```
# program to demonstrate
# range in the character class

# Actual String
$str = "6lpeeks";

# Prints match found if
# it is found in $str
# using range
if ($str =~ /[0-7a-z]/)
```

```

{
    print "Match is Found\n";
}

# Prints match is not found
# if it is not found in $str
else
{
    print "Match is Not Found\n";
}

```

Negated character class: Simply use the caret (^) symbol to negate a character class. It will negate the character supplied following the symbol or even a range. Using caret (^) as the first character in a character class signifies that the character class can match any character except those listed in the character class.

Example:

```

# program to demonstrate
# negated character class

# Actual-String
$str = "peeks56";

# using the negated character class
# Prints match found if
# it is found in $str
if ($str =~ /^[^peeks0-7]/)
{
    print "Match is Found\n";
}

# Prints match is not found
# if it is not found in $str
else
{
    print "Match is Not Found\n";
}

```

SPECIAL CHARACTER CLASSES IN REGULAR EXPRESSIONS

Perl has several distinct character classes, some of which are used so frequently that a specific sequence is built for them. Writing a custom sequence aims to make the code more understandable and concise. Perl's special character classes are as follows:

1. **Digit `\d[0-9]`:** The `d` character is used to match any digit character and is identical to `[0-9]`. A single digit will be matched by the regex `/\d/`. The `d` has been standardized to “digit.” The key advantage is that the user may write in a shorter form that is easier to read. This unique character class can be used in two ways. Let's look at example to understand how to match the character string.

Example:

```
/# [MNOPQ] - \d\d\d\d/
```

The character, as mentioned earlier, the string will match as follows:

```
#M-12345
```

```
#N-66666
```

We may also utilize quantifiers in this case by placing them on the character class.

Example:

```
/# [MNOPQ] - \d{5}/
```

The above example is identical to the previous regex in that it accepts any number of digits following the dash and may be expressed as `/#[MNOPQ]-\d+/.`

In larger character classes, the second technique is utilized. The `\d` is surrounded by square brackets and matches a single character digit.

Example:

```
[\dABCDEFGHIJKLMN]
```

A single digit or any capital letters A, B, C, D, E, F, G, H, I, J, K, L, M, or N can match. It can be written more concisely by using a dash (-). Then it will be something like:

```
[\dA-N]
```

2. **PO SIX character classes:** PO SIX character classes are the standards for ensuring operating system compatibility and establishing the application programming interface (API), including command line shells and utility interfaces. It also provides a number of “character groupings” with names like (alpha, alnum, ascii, blank, etc.). The PO SIX character classes are always in the form of [:class:], where class is the name and the delimiters are [: and:]. POSIX character classes are always contained within the bracketed character classes. These classes provide a quick and easy method to list a set of characters.

Syntax:

```
$string =~ /[[:class:]]/
```

Here, class can be alpha, alnum, ascii, and so on.

As illustrated here, POSIX character classes support larger bracketed character classes:

```
[01[:Class:]]%
```

It will match “0,” “1,” any character classes, and the % sign in this case. Perl supports the following PO SIX character classes, as given in the table below:

Class	Description
Alpha	Any alphabetical character (“[A-Za-z]”)
Alnum	Any alphanumeric character (“[A-Za-z0-9]”)
Ascii	Any character in ASCII character set
blank	A space or horizontal tab
Cntrl	Any control character
Digit	Any decimal digit (“[0-9]”)
graph	Any printable character, excluding space
lower	Any lowercase character (“[a-z]”)
punct	Any graphical character
space	Any whitespace character
upper	Any uppercase character (“[A-Z]”)
xdigit	Any hexadecimal digit (“[0-9a-fA-F]”)
word	Perl extension (“[A-Za-z0-9_]”), equivalent to “\w”

3. **Word character \w[0-9a-zA-Z]:** The letter w belongs to the word character class. The w character can be any single alphanumeric character, including an alphabetic character, a decimal digit, or a

punctuation character such as underscore (`_`). It will only match single character words, not entire words. If we wish to match the entire word, use `\w+`.

- 4. **Whitespace `\s[\t\n\f\r]`:** The character class `\s` will only match one character, a whitespace. It will also match the five letters `\t`-horizontal tab, `\n`-newline, `\f`-form feed, `\r`-carry return, and space. In Perl 5.18, a new character will be introduced corresponding to the `\cK` – vertical tab.
- 5. **Negated character classes `\D`, `\W`, `\S`:** In this universe, there are almost 110, 000 Unicode characters. Just use the caret (`^`) sign to negate a character class. It will negate the character specified following the symbol or even a range. We utilize `[\^d]` in negated character classes to negate digits 0 through 9. However, instead of `[\^d]`, we may just use `\D` to negate the numbers 0 to 9. The table below depicts the special negated character classes:

Character Class	Negated	Meaning	Description
<code>\d</code>	<code>\D</code>	<code>[\^d]</code>	matches to any non-digit character
<code>\s</code>	<code>\S</code>	<code>[\^s]</code>	matches to any non-whitespace character
<code>\w</code>	<code>\W</code>	<code>[\^w]</code>	matches to any non-“word” character

- 6. **Unicode character classes:** Unicode is a definition of “all” existing characters, and the Unicode Standard assigns a unique number to each platform-independent character. There are over 100,000 characters on this planet, and each character is specified as a character point. Some of the characters, however, are grouped.

Syntax:

```
\p{...any character...}
```

This syntax matches a single character from one of the groupings. If we need to match anything other than a specific character, use the matching `\P{...any character...}` expression.

QUANTIFIERS IN REGULAR EXPRESSION

Perl has several regular expression quantifiers that may determine how many times a particular character can repeat before it is matched. This is mainly utilized when the number of characters to match is uncertain.

Perl quantifiers are classified into six types, which are listed below:

1. * = This indicates that the component must appear zero or more times.
2. + = This indicates that the component must appear one or more times.
3. ? = This specifies that the component must appear either zero or once.
4. {n} = This specifies that the component must be present n times.
5. {n,} = This specifies that the component must appear at least n times.
6. {n, m} = This specifies that the component must appear at least n times and no more than m times.

Quantifier Table

All of the above kinds may be comprehended using this table, which contains regular expression quantifiers and examples.

Regex	Examples
/Bx*B/	BB, BxB, BxxB, BxxxB,
/Bx+B/	BxB, BxxB, BxxxB,
/Bx?B/	BB, BxB
/Bx{1, 3}B/	BxB, BxxB, BxxxB
/Bx{2, }B/	BxxB, BxxxB,
/Bx{4}B/	BxxxxB

Let's look at some examples of these quantifiers.

First example: In this example, the * quantifier is employed in a regular expression /Pe*ks/, which yields either “Pks,” “Peks,” “Peeks”...and so on, and is matched with the input text “Pks,” yielding “Match Found.”

```
#!/usr/bin/perl

# Initializing string
$a = "Pks";

# matching above string with "*"
# quantifier in the regular expression /Ge*ks/
if ($a =~ m/Pe*ks/)
{
    print "Match is Found\n";
}
```

```

else
{
    print "Match is Not Found\n";
}

```

Second example: In this example, the + quantifier is employed in a regular expression /Pe+ks/, which yields either “Peks,” “Peeks,” “Peeks” ...and so on, and is matched with the input text “Pks,” yielding “Match Not Found.”

```

#!/usr/bin/perl

# Initializing string
$a = "Pks";

# matching the above string with "+"
# quantifier in the regular expression /Pe+ks/
if ($a =~ m/Pe+ks/)
{
    print "Match is Found\n";
}
else
{
    print "Match is Not Found\n";
}

```

Third example: In this example, the ? quantifier is used in a regular expression /Pe?ks/, which yields either “Peks” or “Pks,” and is matched with the input text “Peks,” yielding “Match is Not Found.”

```

#!/usr/bin/perl

# Initializing string
$a = "Peeks";

# matching the above string with "?"
# quantifier in the regular expression /Pe?ks/
if ($a =~ m/Pe?ks/)
{
    print "Match is Found\n";
}
else

```

```
{
    print "Match is Not Found\n";
}
```

Fourth example: In this example, the {n} quantifier is employed in a regular expression `Pe{2}ks`, which yields “Peeks,” and is matched with an input text “Peeks,” yielding “Match is Found.”

```
#!/usr/bin/perl

# Initializing string
$a = "Peeks";

# matching the above string with {n}
# quantifier in the regular expression /Pe{2}ks/
if ($a =~ m/Pe{2}ks/)
{
    print "Match is Found\n";
}
else
{
    print "Match is Not Found\n";
}
```

Fifth example: In this example, the {n, } quantifier is used in a regular expression `/Pe{2, }ks/`, which creates “Peeks,” “Peeeks,” “Peeeeeks,” and so on, and when matched with the input text “Peks,” it returns “Match is Not Found.”

```
#!/usr/bin/perl

# Initializing string
$a = "Peks";

# matching the above string with {n, }
# quantifier in the regular expression /Pe{2, }ks/
if ($a =~ m/Pe{2, }ks/)
{
    print "Match is Found\n";
}
else
{
    print "Match is Not Found\n";
}
```

Sixth example: In this example, the {n, m} quantifier is employed in a regular expression `/Pe{1, 2}ks/`, which yields “Peks” and “Peeks,” and is matched with an input text “Peeks,” yielding “Match is Not Found.”

```
#!/usr/bin/perl

# Initializing string
$a = "Peeeks";

# matching the above string with {n, m}
# quantifier in then regular expression /Pe{1, 2}ks/
if ($a =~ m/Pe{1, 2}ks/)
{
    print "Match is Found\n";
}
else
{
    print "Match is Not Found\n";
}
```

BACKTRACKING IN REGULAR EXPRESSION

A regular expression (a.k.a. regexes, regexps, or REs) in Perl is a means of representing a group of strings without having to describe all strings in your program. It is essentially a series of characters used for pattern matching. Regex have several uses in Perl:

- To begin with, they are used in conditionals to assess if a string matches a specific pattern.

For example: Regex in conditionals

```
#!/usr/bin/perl

# Regular expressions in the Conditionals
# Program to determine whether string
# matches particular pattern
print "How are you feeling?\n";
my $stmt = <>;
print($stmt);
if ($stmt == /hungry/)
```

```
{
    print "\n What do we wish to have?\n";
    my $ip = <>;
    print($ip);
}
```

Here, the user's input is matched; if we have the word “hungry,” i.e., if condition is true, it will print “What do we wish to have?” Otherwise, it will go to the following condition or statement.

- Second, they may find patterns in a string and replace them with something else.

For example: Substitution operator

```
#!/usr/bin/perl
# Regular expressions in the Substitutions

# Program to determine whether string
# matches particular pattern and replaces it
print "Whats your thought on life\n";
my $stmt = <>;
print ($stmt);

# Substitution using the regex
$stmt =~ s/worst/good/;
print ("\n$stmt");
```

The preceding code uses “good” instead of “worst.”

- Finally, patterns can describe not just where something exists but also where it does not exist. As a result, the split operator employs a regular expression to identify where the data does not exist. In other words, the regular expression specifies the separators that separate data fields.

For example: Split operator

```
#!/usr/bin/perl

# Program to illustrate the
```

```
# use of split function
$var1 = "Birth";
$var2 = "Life";
$var3 = "Death";

# Using split function
my ($var1, $var2, $var3) = split(/, /, "sab,
mohmaya, hai");
print($var1);
print($var2);
print($var3);
```

The `split` function matches on single comma character in the preceding example.

BACKTRACKING

Backtracking is another key aspect of regular expression matching, which is now employed (when needed) by all regular non-possessive expression quantifiers, namely “*”, “?”, “+”, “+?”, {n, m}, and {n, m}?. Backtracking is frequently optimized internally, but the overall notion above is correct (it returns from a failed recursion on a tree of possibilities). When Perl attempts to match patterns with a regular expression and its previous attempts fail, or when the matching patterns are saved for future use, it backtracks.

For instance, `/.?/` may be used to match anything comparable to an HTML tag such as “Bold.” This forces the pattern’s two parts to match the same string, in this instance “B.”

Consider another example:

```
/^ab*bc*d/
```

The preceding regexp can be interpreted as follows:

1. Begin at the start of the string.
2. Match an “a.”
3. Match as many “b”s as possible, although failing to match any is OK.
4. Match as many “c”s as possible, although failing to match any is OK.
5. Match as many “d”s as we can, although not all are required.

“e” MODIFIER IN REGULAR EXPRESSION

The regular expression in Perl allows us to execute numerous actions on a given text using appropriate operators. These operators can execute operations such as string modification, substituting other substrings, and so on. Substitution of a substring in a given string is accomplished with the “s” (substitution) operator, which accepts two operands: the substring to be replaced and the replacement string.

```
s/To_be_replaced/Replacement/
```

Furthermore, the “e” modifier is used to replace the substring with a replacement string that is a regular expression to be evaluated. The “e” modifier is added to the substitution expression.

```
s/To_be_replaced/Regular_Expression/e;
```

The “e” modifier can also use with the “g” (globally) modifier to affect all possible substrings in the specified string.

First example: Substitution using a character class

```
#!/usr/bin/perl

# Defining string to be converted
$String = "Peeks for Peaks is the best";
print "The Original String: $String\n";

# Converting string to UPPERCASE
# using 'uc' Function
$String =~ s/(\w+)/uc($1)/ge;
print "Uppercased String: $String\n";

# Converting string to lowercase
# using the 'lc' Function
$String =~ s/(\w+)/lc($1)/ge;
print "Lowercased String: $String\n";
```

The above code uses the character class “w,” which contains the lower and upper case alphabets and all digits (a-z|A_Z|0-9). This is used to replace the entire string with a single replacement operation.

Second example: Use a single letter or word for a particular substitution.

```
#!/user/bin/perl

# Defining string to be converted
$string = "Peeks for Peeks is the best";
print "Original String: $string\n";

# Converting single character using e modifier
$string =~ s/(e)/uc($1)/ge;
print "Updated String: $string\n";

# Converting word using e modifier
$string =~ s/(for)/uc($1)/ge;
print "The Updated String: $string\n";
```

The string after updating will not revert to its original value, even after the second recursion, as seen in the preceding code.

The Substitution Operation Is Performed using a Subroutine

Subroutine substitution in Perl regex may also be done using subroutines to eliminate the repetition of writing the substitution regex for each string. This is accomplished by inserting the regex code in the subroutine and invoking it as needed.

Example:

```
#!/usr/bin/perl

# Subroutine for the substitution operation
sub subroutine
{
    $regex = shift;
    $regex =~ s/Friday/Tuesday/;
    return $regex;
}

# Defining string to be converted
$string = "Monday Friday Wednesday";
print "Original String: $string\n";
```

```
# Calling subroutine for substitution
$string =~ s/(\w+)/subroutine($1)/ge;
print "Updated String: $string\n";

# Defining new String to be converted
$string2 = "Today is Friday";
print "\nThe Original String: $string2\n";

# Calling subroutine for substitution
$string2 =~ s/(\w+)/subroutine($1)/ge;
print "The Updated String: $string2\n";
```

When the substitution operation begins in the preceding code, it calls the subroutine “change_substitution,” which contains the regex code for changing the substring that matches the search.

REGEX “ee” MODIFIER

The regular expression in Perl allows you to execute numerous actions on a given text using appropriate operators. These operators can execute operations such as string modification, substituting other substrings, and so on. Substitution of a substring in a given string is accomplished with the “s” (substitution) operator, which accepts two operands: the substring to be replaced and the replacement text.

```
s/To_be_replaced/Replacement/
```

In Perl, modifiers match a string with a specified pattern using a regular expression. In Perl, the “ee” modifier is equivalent to the “\e” modifier. It is used to evaluate the string on the right and then to assess the result further. In Perl, it is equivalent to the double “eval” operator. The “\e” operator is used to evaluate the string on the right. “\ee” is one step ahead of it. It uses the “\e” operator on a string that already has the “e” operator.

```
s///ee;
```

The “ee” modifier, like the “e” modifier, can be combined with the “g” (globally) modifier to apply modifications to all possible substrings in the provided string.

Example:

```
#!/usr/bin/perl
my $var = 'for';
```

```
# Defining string
my $String = 'Peeks $var Peeks is the best';

# String before using the 'ee' modifier
print "The Original String: $String\n";

# Applying 'ee' modifier using the regex
$String =~ s/(\$\w+)/$1/ee;
print "The Updated String: $String";
```

Because `$var` is considered a substring of the provided string and is not considered a variable, it is written as it was before the regex was applied to the string in the preceding code. However, once the regex is applied, the “ee” modifier evaluates the value of `$var` and displays it.

If we apply the “e” modifier in the above code, the resultant string is the same as the original string:

```
#!/usr/bin/perl
my $var = 'for';

# Defining string
my $String = 'Peeks $var Peeks is the best';

# String before using 'e' modifier
print "The Original String: $String\n";

# Applying 'e' modifier using regex
$String =~ s/(\$\w+)/$1/e;
print "The Updated String: $String";
```

This is because when the “e” modifier is applied, the regex considers the RHS to be the evaluated string to be used as a replacement; in this case, the RHS is `$1`, which retains `$var` but not its value. As a result, applying the “ee” modifier will re-evaluate the RHS, which contains the already eval’d value `$var`.

When Doing Mathematical Calculations, Use the “e” Modifier

If a mathematical expression is contained in a string, the value of the expression is not evaluated and is displayed as is. This is because it is seen as both a string and an expression to be evaluated.

Example:

```
#!/usr/bin/perl

# Mathematical-expression
# stored as a string
$String = "1 + 2";

# Regex to evaluate sum
$String =~ s/(\d+ [+*\/-] \d+)/$1/ee;

print "Sum is $String";
```

The expression in the above code is written in Regex, using the “d+” operator for writing one or more digits, the “[+*\/-]” character class for the operator symbol, and then another “d+” for the digit. The “ee” modifier evaluates the string and returns the expression’s total, which is displayed with the \$1 operator.

The above regex may also be kept in a subroutine and used to evaluate different expressions without having to rewrite the regex.

Example:

```
#!/usr/bin/perl

# Subroutine to calculate the regex
sub Regex
{
    $var = shift;

    # Regex to evaluate sum
    $var =~ s/(\d+ [+*\/-] \d+)/$1/ee;
    return $var;
}

# The Mathematical expression
# stored as string
$String1 = "1 + 2";

$String1 = Regex($String1);

print "Sum is $String1\n";
```

```
# Calculating-product
$string2 = "10 * 3";
$string2 = Regexp($string2);

print "Product is $string2";
```

A single regex can be used in the given code to execute four mathematical calculations utilizing the subroutine.

pos() FUNCTION IN REGULAR EXPRESSION

The pos() function in Perl returns the position of the last match in Regexp using the “m” modifier.

The pos() function in Regexp can be used in conjunction with the character classes to return list of all the required substring positions in given string. To search for a substring within the entire text, use the global operator “g” in conjunction with the “m” modifier.

```
Syntax: pos(String)
Parameter: String after applying the Regular
Expression
Returns: position of the matched substring
```

First example: Making use of a substring character

```
#!/usr/bin/perl

# Program to print position of substring
$string = "Peeks For Peeks";

print" Position of 'P' in string:\n";

# Regexp to search for the substring
# using the m modifier
while($string =~ m/G/g)
{

    # Finding position of substring
    # using pos() function
    $position = pos($string);
    print "$position\n";
}
```

Second example: Making use of a character class

```
#!/usr/bin/perl

# Program to print position of substring
$String = "Peeks For Peeks";

print "Position of all the Uppercase characters:\n";

# Regex to search for
# all upper case characters
# using character class
while($String =~ m/[A-Z]/g)
{

    # Finding position of substring
    # using pos() function
    $position = pos($String);
    print "$position, ";

}

print "\nThe Position of all Lowercase characters:\n";

# Regex to search for
# all the lower case characters
# using the character class
while($String =~ m/[a-z]/g)
{

    # Finding position of substring
    # using the pos() function
    $position = pos($String);
    print "$position, ";

}
```

Third example: Position of the spaces

```
#!/usr/bin/perl

# Program to print position of substring
$String = "Peeks For Peeks";
```

```
# Regex to search for
# all spaces
while($String =~ m/\s/g)
{

    # Finding position of substring
    # using the pos() function
    $position = pos($String);
    print "$position\n";
}
```

To Match from a Specified Position, use \G Assertion

In Perl Regex, the \G Assertion is used to match a substring beginning at a position specified by the pos() function and concluding at the matching character specified in the regex. The position of the first occurrence of the character specified by the “m” modifier will return.

Example:

```
#!/usr/bin/perl

# Defining default string
$_ = "Peeks World is the best";

# Terminating character
# using the m modifier
m/o/g;

# Specifying starting position
$position = pos();

# Using the \G Assertion
m/\G(.*)/g;

# Printing position
# and the remaining string
print "$position $1";
```

In the preceding example, the position of the first occurrence of the matching substring is printed alongside the remaining string. If we need to restart the counting position for the next occurrence of the matching character, simply store the remaining string in \$1 into the default string.

Example:

```
#!/usr/bin/perl

# Defining default string
$_ = "Peeks World is the best among all";

# Terminating character
# using the m modifier
m/o/g;

# Specifying starting position
$position = pos();

# Using the \G Assertion
m/\G(.*)/g;

# Printing position
# and remaining string
print "$position $1\n";

# To start counting from matched character
# until next possible match
$_ = $1;
m/o/g;

$position = pos();

# Using the \G Assertion
m/\G(.*)/g;

# Printing position
# and remaining string
print "$position $1\n";
```

REGEX CHEAT SHEET

Regex are an essential part of Perl programming. It is used to look for the specified text pattern. A group of characters forms the search pattern. It is also referred to as regexp. When a user learns Regex, he may require a quick review of concepts not frequently used. A regex cheat sheet containing the various classes, characters, modifiers, and so on that are used in the regular expression is created to provide that service.

Character Classes

To match the string of characters, character classes are used. These classes allow the user to match any range of characters that the user does not know in advance.

Classes	Explanation
[abc.]	It includes only one of the specified characters i.e. “a,” “b,” “c,” or “.”
[a-j]	It includes all characters from a to j.
[a-z]	It includes all the lowercase characters from a to z.
[^az]	It includes all the characters except a and z.
\w	It includes all the characters like [a-z, A-Z, 0-9].
\d	It matches for digits like [0-9].
[ab][^cde]	It matches that characters a and b should not be followed by c, d, and e.
\s	It matches for the [\f\t\n\r] i.e. form feed, tab, newline, and carriage return.
\W	Complement of the \w.
\D	Complement of the \d.
\S	Complement of the \s.

Example:

```
# Perl program to demonstrate
# character class

# Actual String
$str = "45char";

# Prints match found if
# its found in $str
# by using \w
if ($str =~ /\w/)
{
    print "Match Found\n";
}

# Prints match not found
# if it is not found in $str
else
{
    print "Match Not Found\n";
}
```

Anchors

Anchors do not correspond to any character. Instead, they correspond to a specific position such as before, after, or between the characters.

Anchors	Explanation
^	It matches the beginning of string.
\$	It matches at the end of string.
\b	It matches the word boundary of string from \w to \W.
\A	It matches the beginning of string.
\Z	It matches at the ending of string or before the newline.
\z	It matches only at the end of string.
\G	It matches at specified position pos().
\p{...}	The Unicode character class like IsLower, IsAlpha, etc.
\P{...}	Complement of the Unicode character class.
[:class:]	The POSIX Character Classes like digit, lower, ascii, etc.

Example:

```
# program to demonstrate
# use of anchors in the regex

# Actual String
$str = "55";

# Prints match is found if
# it is found in $str
# using the Anchors /
if ($str =~ /[[:alpha:]]/)
{
    print "Match is Found\n";
}

# Prints match not found
# if its not found in $str
else
{
    print "Match is Not Found\n";
}
```

Metacharacters

Metacharacters are used in Perl Regex to match patterns. All metacharacters must be avoided.

Characters	Explanation
^	To check the beginning of a string.
\$	To check the ending of the string.
.	Any character except newline.
*	Matches 0 or more times.
+	Matches 1 or more times.
?	Matches 0 or more times.
()	Used for the grouping.
\	Use for the quote or special characters.
[]	Used for the set of characters.
{}	Used as the repetition modifier.

Quantifiers

These are used to detect special characters. Quantifiers are classified into three types:

- “?” It matches for 0 or 1 character occurrence.
- “+” It corresponds to one or more occurrences of the character.
- “*” It matches when the character appears 0 or more times.

Using quantifiers	Explanation
a?	Checks whether “a” occurs 0 or 1 time.
a+	Checks whether “a” occurs 1 or more times.
a*	Checks whether “a” occurs 0 or more times.
a{2, 6}	Checks whether “a” occurs 2 to 6 times.
a{2, }	Checks whether “a” occurs 2 to infinite times.
a{2}	Checks whether “a” occurs 2 times.

Example:

```
# program to demonstrate
# use of quantifiers in the regex

# Actual String
$str = "color";
```

```
# Prints match is found if
# it is found in $str
# using quantifier?
if ($str =~ /colou?r/)
{
    print "Match is Found\n";
}

# Prints match is not found
# if it is not found in $str
else
{
    print "Match is Not Found\n";
}
```

Modifiers

Modifiers	Explanation
\g	It is used to replace all the occurrences of string.
\gc	It allows continued search after the \g match fails.
\s	It treats the string as a single line.
i	It turns off case sensitivity.
\x	It disregards all-white spaces.
(?#text)	It is used to add comments to the code.
(?:pattern)	It is used to match the pattern of the non-capturing group.
(? pattern)	It is used to match the pattern of the branch test.
(?=pattern)	It is used for the positive look ahead assertion.
(?!pattern)	It is used for the negative look ahead assertion.
(<=pattern)	It is used for the positive look behind the assertion.
(<!pattern)	It is used for the negative look behind the assertion.

White Space Modifiers

Modifiers	Explanation
\t	Used for the inserting tab space.
\r	Carriage return character.
\n	Used for inserting a new line.
\h	Used for the inserting horizontal white space.
\v	Used for the inserting vertical white space.
\L	Used for the lowercase characters.
\U	Used for the upper case characters.

Quantifiers – Modifiers

Maximal	Minimal	Explanation
?	??	It can happen 0 or 1 time.
+	+?	It can happen one or more times.
*	*?	It could happen 0 or more times.
{3}	{3}?	It must match exactly three times.
{3, }	{3, }?	At least three times must match.
{3, 7}	{3, 7}?	Must match at least three times and no more than seven times.

Grouping and Capturing

Inside regex, these groups are denoted by “1,” while outside regex, they are denoted by “\$1.” This is known as capture. These groups can fetch by variable assignment in a list context. The grouping construct (...) generates capture buffers and groups of captures.

(...)	These are used for the grouping and capturing.
\1, \2, \3	During the regex matching, these are used to capture buffers.
\$1, \$2, \$3	During the successful matching, these are used to capture the variables.
(?...)	These are used to group items without capturing them. (Neither of these set this \$1 nor 1).

SEARCHING IN A FILE USING REGEX

In Perl, a regular expression (Regex, Regexp, or RE) is a special text string that describes a search pattern within a given text. Regex in Perl is tied to the host language and is not the same as in PHP, Python, etc. These are sometimes referred to as “Perl 5 Compatible Regular Expressions.” Binding operators such as = ~ (Regex Operator) and !~ (Negated Regex Operator) are used to use the Regex.

Binding regex operators are used to matching a string against a regular expression. The statement’s left side will contain a string that matches the specified pattern on the right side. The negated regex operator determines whether or not the string is equal to the regular expression specified on the right-hand side.

Regex operators aid in searching a file for a specific word or group of words. This can accomplish in a variety of ways, depending on the user’s specific requirements. Perl searches follow the standard format of opening the file in read mode and then reading the file line by line, looking for required string or group of strings in each line. When the required match

is found, the statement following the search expression determines what to do with the matched string, which can add to any other file specified by the user or printed on the console.

There are several ways to look for the required string within the regular expression that was created to match the string with the file.

Regular Search

This basic pattern for writing a regular expression searches for the specified string within the specified file. The syntax of such a regular expression is as follows:

```
$String =~ /the/
```

This expression will look for lines in the file that contain the letters “the” and store that word in the variable \$String. Furthermore, the value of this variable can be copied to a file or printed to the console.

Example:

```
use strict;
use warnings;

sub main
{
    my $file = 'C:\Users\PeeksForPeeks\PEP.txt';
    open(FH, $file) or die("File $file is not
found");

    while(my $String = <FH>)
    {
        if($String =~ /the/)
        {
            print "$String \n";
        }
    }
    close(FH);
}
main();
```

In the list of words that contain the word “the,” to avoid such words, the regular expression can be modified as follows:

```
$String =~ / the /
```

By inserting spaces before and after the required word to be searched, the searched word is isolated from both ends, and no such word containing it as a part of it is returned in the search process. This will eliminate the need to search for unnecessary words. However, this will exclude words that contain a comma or a full stop immediately after the requested search word.

To avoid this situation, there are other methods for limiting the search to a specific word, one of which is using the word boundary.

Using Word Boundary in the Regex Search

As seen in the preceding example, a regular search returns either the extra words that include the searched word or excludes some of the words if the required word is preceded and followed by spaces. A word boundary, denoted by the letter “\b,” is used to avoid this.

```
$String =~ /\bthe\b/;
```

This will limit the words that contain the requested word as part but will not exclude words that end with comma or full stop.

Example:

```
use strict;
use warnings;

sub main
{
    my $file = 'C:\Users\PeeksForPeeks\PEP.txt';
    open(FH, $file) or die("File $file is not
found");

    while(my $String = <FH>)
    {
        if($String =~ /\bthe\b/)
        {
            print "$String \n";
        }
    }
    close(FH);
}
main();
```

The word that ends with a full stop is included in the search, but words that contain the searched words as a part are not. As a result, word boundary can aid in overcoming the issue created by the regular search method.

What if there is a need to find words that begin, end, or both begin and end with specific characters? That cannot be accomplished using regular search or the word boundary. Perl allows the use of wildcards in Regex in scenarios like these.

Use of Wildcards in the Regular Expression

With wildcards in regular expression, Perl allows us to search for a specific set of words or words that follow a specific pattern in a given file. Wildcards are “dots” placed within the regex in addition to the required word to be searched. These wildcards allow the regex to search for and display all related words that match the given pattern. Wildcards help reduce the number of iterations required to search for different words that share a pattern of letters.

```
$String =~ /t..s/;
```

The above pattern will look for words that begin with t, end with s, and have two letters/characters between them.

Example:

```
use strict;
use warnings;

sub main
{
    my $file = 'C:\Users\PeeksForPeeks\PEP.txt';
    open(FH, $file) or die("File $file is not
found");

    while(my $String = <FH>)
    {
        if($String =~ /t..s/)
        {
            print "$String \n";
        }
    }
    close(FH);
}
main();
```

The preceding code contains all of the words specified in the given pattern.

This method of printing the searched words prints the entire line that contains that word, making it difficult to determine exactly what word the user is looking for. To avoid confusion, we can only print the searched words rather than the entire sentence. This is accomplished by grouping the searched pattern with parentheses. \$number variables are used to print this grouping of words.

The \$number variables are matches from the regular expression's last successful match of the capture groups. For example, if the regular expression contains multiple groupings, \$1 will print the words that match the first grouping, \$2 will print the words that match the second grouping, and so on.

The following program has been modified using the \$number variables to display only the searched words rather than the entire sentence:

```
use strict;
use warnings;

sub main
{
    my $file = 'C:\Users\PeeksForPeeks\PFP.txt';
    open(FH, $file) or die("File $file is not found");

    while(my $String = <FH>)
    {
        if($String =~ /(t..s)/)
        {
            print "$1 \n";
        }
    }
    close(FH);
}
main();
```

This chapter covered Regex, operators, regex character and special character classes, and quantifiers and backtracking. Moreover, we discussed “e” modifier in regex, “ee” Modifier, and pos() function.

Object-Oriented Programming in Perl

IN THIS CHAPTER

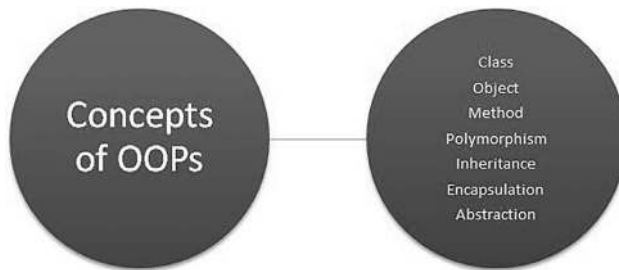
- Classes and Objects
- Methods
- Constructors and Destructors
- Method Overriding
- Inheritance
- Polymorphism
- Encapsulation

In the previous chapter, we discussed regular expressions, and in this chapter, we will cover object-oriented programming.

Object-oriented programming (OOPs): As the name implies, OOPs refers to programming languages that use objects. OOP aims to implement real-world entities in programming, such as inheritance, hiding, and polymorphism. The primary goal of OOP is to connect the data and the functions that operate on it so that no other part of the code can access the data except that function.

Concepts of OOPs:

- Class
- Object
- Method
- Polymorphism
- Inheritance
- Encapsulation
- Abstraction



Concepts of OOP's.

Let us look at the various features of an OOPs language:

1. **Class:** A class is a user-defined blueprint or prototype used to create objects. It represents a set of properties or methods shared by all objects of the same type. In general, class declarations can include the following components in the following order:
 - Class name: The name of the class should begin with an initial letter (capitalized by convention).
 - Superclass (if any): If applicable, the name of the class's parent (superclass), preceded by the keyword "use."
 - Constructors (if any): Perl subroutine constructors return an object that is an instance of the class. The constructor is usually named "new" in Perl.
 - Body: The class body is surrounded by braces {}.

2. **Object:** It is a fundamental unit of OOPs that represents real-world entities. A typical Perl program generates many objects, which interact by invoking methods. An object is made up of:

- **State:** State is represented by an object's attributes. It also reflects an object's properties.
- **Behavior:** Behavior is represented by an object's methods. It also reflects an object's interaction with other objects.
- **Identity:** It gives an object a unique name and allows one object to interact with other objects.

Example: A dog is an example of an object.

3. **Method:** A method is a group of statements that perform a specific task and return the result to the caller. A method can complete a task without returning anything. Methods save time by allowing us to reuse code without having to retype it.

4. **Polymorphism:** Polymorphism refers to the ability of OOPs programming languages to distinguish between entities with the same name efficiently. Perl accomplishes this through the use of these entities' signatures and declarations.

Polymorphism in Perl is primarily of two types:

- Perl overloading
- Perl overriding

5. **Inheritance:** An essential pillar of OOP is inheritance (object-oriented programming). It is the mechanism in Perl that allows one class to inherit the features (fields and methods) of another.

Important terms to remember:

- **Superclass:** A superclass is a class with inherited characteristics (a base class or a parent class).
- **Subclass:** It is a class that derives from the another (or derived class, extended class, or child class). The subclass can add its fields and methods to those of the superclass.
- **Reusability:** Inheritance promotes "reusability," which means that if we want to create new class and there is already class that includes some of the code we want, we can derive our new class

from the existing class. By doing so, we are reusing the existing class' fields and methods.

A class in Perl can be created with packages and inherited with the “use” keyword.

Syntax:

```
use packagename
```

6. **Encapsulation:** Encapsulation is wrapping data into a single unit. It is the mechanism that connects code and the data that it manipulates. Encapsulation can also be considered a protective shield that prevents code from accessing data outside of the shield.

- Technically, in encapsulation, a class' variables or data are hidden from other classes and can only be accessed through any member function of the class in which they are declared.
- Because the data in a class is hidden from other classes, it is also referred to as data-hiding.
- Encapsulation can accomplish by declaring all variables in the class as private and writing public methods in the class to set and retrieve variable values.

7. **Abstraction:** Data abstraction is the property that only displays the essential details to the user. The user is not shown the trivial or non-essential units. For example, a car is considered a whole rather than its individual components.

Data abstraction is also defined as the process of identifying only the necessary characteristics of an object while ignoring irrelevant details. An object's properties and behaviors distinguish it from other objects of a similar type and aid in classifying/grouping the objects.

Consider the following scenario: A man is driving a car. The man only knows that pressing the accelerators will increase the car's speed or that applying the brakes will stop the car. Still, he has no idea how pressing the accelerator will increase the speed, nor does he understand the car's inner mechanism or how the car's accelerator, brakes, and other controls are implemented. This is the definition of abstraction.

CLASSES IN OOP

In today's world, when programming is used in every aspect of our life, we must adopt programming paradigms that are closely related to real-world instances. The competitiveness and complexity of real-world challenges have undergone a significant transformation, necessitating the need for more adaptable ways in the industry.

The notion of OOPs is a wonderful method to shape any programming language such that solving real-world cases is a breeze. OOP is a programming paradigm that places more emphasis on objects than procedures. The OOPs idea involves working in terms of objects and the interaction of objects.

Note: OOP seeks to implement real-world elements, such as inheritance, encapsulation, and polymorphism. The primary goal of OOP is to connect the data and the functions that act on them such that no other portion of the code may access the data than the function.

Object

It is an instance of a class inside a data structure with specific properties and behavior. An "Apple" is an example of an object. It has the qualities of being a fruit with a crimson hue, etc. Its conduct is "it tastes delicious."

Its data represent the features of an object, and its activity is represented by its related functions, according to the OOPs idea. Thus, an object is an entity that holds data and provides a function-based interface.

Class

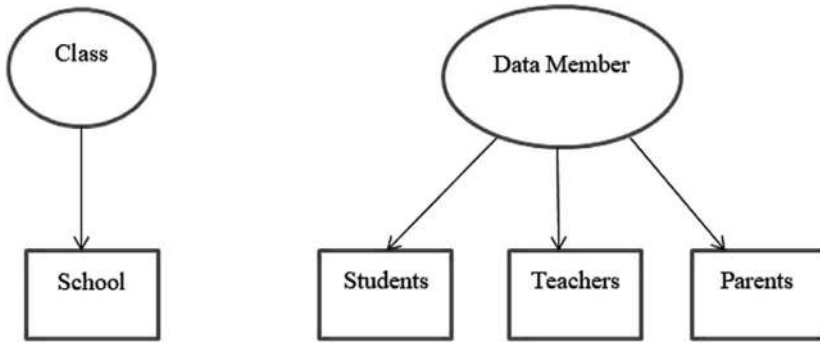
A class is an enlarged data structure idea. It specifies the prototype blueprint of data-containing objects.

Objects are instances of their respective classes. A class consists of data members and data functions; it is a user-defined, pre-defined data type that can be accessed and used by creating an instance of that class.

Example: Consider a class at school as an example. There may be schools with various names and organizational structures, but they all have some qualities, such as students, instructors, and staff. Therefore, school is the class containing the data members, teachers, students, and parents, and the member functions may be `calculate_students_marks()`, `calculate_teachers_salary()`, and `Parents_Database ()`.

Data Member

Data members are data variables, and member functions are the functions used to control these variables; these data members and member functions combined determine the attributes and behavior of the objects inside a class.



Data member.

Defining a Class

Perl makes it very simple to define a class. A class in Perl corresponds to a package. To define a class, we must first load and construct a Package. A Package is a pre-contained unit of user-defined variables and subroutines that can use anywhere in the program.

Syntax:

```
package Classname
```

Creating a Class and Making Use of Objects

A class can create in Perl using the keyword package, but an object is created by calling a constructor. A constructor is defined as a method in a class.

Creating a Class Instance

The class and constructor names can be whatever the user wants. Most programmers prefer “new” as a constructor name for their programs because it is easier to remember and use than any other complex constructor name.

```
package student // This is class student
sub Student_data // Constructor to the class
```

```

{
    my $class = shift;
    my $self = {
        _StudentFirstName => shift;
        _StudentLastName => shift;
    };

    print "The Student's First Name is $self
->{_StudentFirstName}\n";
    print "The Student's Last Name is $self
->{_StudentLastName}\n";
    bless $self, $class;
    return $self;
}

```

A function called `bless` is used in the preceding example code. This function is used to associate an object with a class passed as an argument.

Syntax:

```
bless Objectname, ClassName
```

Creating an Object

In Perl, an object is created by calling the constructor defined in the class. An object name can be any variable the user requires, but it is customary to name it in relation to the class.

```
$Data = Student_data student( "Diksha", "Mayank");
```

Example:

```

use strict;
use warnings;

package student;

# constructor
sub student_data
{

    # shift will take package name 'student'
    # and assign it to the variable 'class'

```

```

my $class_name = shift;
my $self = {
    'StudentFirstName' => shift,
    'StudentLastName' => shift
};
# Using the bless function
bless $self, $class_name;

# returning object from the constructor
return $self;
}

# Object creation and constructor calling
my $Data = new student_data student("Geeks",
"forGeeks");

# Printing the data
print "$Data->{'StudentFirstName'}\n";
print "$Data->{'StudentLastName'}\n";

```

Using classes in OOP is critical because it precisely depicts real-world applications and can be used to solve real-world problems.

OBJECTS IN OOPs

Perl is an object-oriented, interpreter-based, dynamic programming language. OOP has three primary components: objects, classes, and methods. An object is a data type that may be referred to as an instance of the class it belongs. It may be a collection of data variables of various data kinds and a collection of various data structures. Methods are functions that operate on class object instances.

The following example illustrates how objects may use in Perl:

We must first define the class. In Perl, this is accomplished by constructing the class' package. A package is an encapsulated object that contains all of the class' data members and functions.

```
package Employee;
```

Employee is the class name in this case.

The second task is to create a package instance (i.e., the object). We'll need a constructor for this. In Perl, a constructor is a subroutine that is usually called "new." However, because the name is user-defined, it is not limited to "new."

```
package Employee;

# Constructor with the name new
sub new
{
    my $class = shift;
    my $self = {
        _serialNum => shift,
        _firstName => shift,
        _lastName => shift,
    };

    bless $self, $class;
    return $self;
}
```

To design our object, we define a basic hash reference `$self` in the constructor. Here, the object will contain three properties for an `Employee`: `serialNum`, `firstName`, and `lastName`. This indicates that each employee will have their serial number, firstname, and lastname. The `my` keyword functions as an access specifier that localizes `$class` and `$self` to the contained block.

The `shift` keyword transfers the package name from the default array `"@"` to the `bless` function. The `bless` method returns a reference that eventually becomes an object. And lastly, the constructor will return an instance of the `Employee` class. The most important aspect is how to initialize an object. It may be accomplished as follows:

```
$object = new Employee(1, "Peeks", "forPeeks");
```

In this case, `$object` is a scalar variable that refers to the hash defined in the constructor.

The following is an example program for creating and implementing objects in OOPs:

```
use strict;
use warnings;

# class with the name Employee
package Employee;

# constructor with name new
sub new
```

```

{
    # shift will take package name
    # and assign it to the variable 'class'
    my $class = shift;

    # defining hash reference
    my $self = {
        _serialNum => shift,
        _firstName => shift,
        _lastName => shift,
    };

    # Attaching object with the class
    bless $self, $class;

    # returning the instance of class Employee
    return $self;
}

# Object creation of class
my $object = new Employee(1, "Peeks", "forPeeks");

# object here is a hash to reference
print("$object->{_firstName} \n");
print("$object->{_serialNum} \n");

```

An object in Perl functions in the same way that it does in other languages, such as C++ and Java. The above program demonstrates the procedure of an object in Perl, including its creation and use in a class.

METHODS IN OOPs

Methods are used to access and alter an object's data. These are the entities that may be called via class or package objects. Methods in Perl are just subroutines; they have no unique identity. A method's syntax is identical to that of a subroutine. Methods, like subroutines, are specified using the sub keyword. As its first parameter, the method accepts an object or the package on which it is called.

OOPs employ these methods to change the object's data and not to interact with the object directly; this is done to ensure the data's security by preventing the programmer from directly altering the object's data.

Various helper methods that accept the object as an argument and save its value in another variable make this possible. Additionally, adjustments are made to the second variable. These changes do not affect the object's data, making it more secure.

Types of Methods in Perl

Based on the inputs given, methods in Perl may be divided into two categories: static and virtual methods.

A static method is one where the method's first parameter is the class name. The functionality of a static method is applied to the whole class since it accepts the class' name as an argument. These are also known as class methods. Since most methods belong to the same class, it is unnecessary to supply the class name as an argument. Example: A class' constructors are considered static methods.

Virtual method is one in which the object reference is supplied as the function's first parameter. The first parameter of a virtual function is relocated to a local variable and then utilized as a reference.

Example:

```
sub Student_data
{
    my $self = shift;

    # Calculating-result
    my $result = $self->{'Marks_obtained'} /
                  $self->{'Total_marks'};

    print "Marks scored by student are: $result";
}
```

In OOP, methods require parentheses to hold the arguments, and these methods are invoked with an arrow operator (->).

get-set Methods

Methods are used to secure an object's data and are thus used with either the object's reference or the value is stored in another variable and then used. In OOPs, get-set methods are used to provide data security to objects. The get-method is used to retrieve the object's current value, while the set value method assigns a new value to the object.

Example:

```
# Declaration and definition of the Base class
use strict;
use warnings;

# Creation of parent class
package vehicle;

# Setter method
sub set_mileage
{
    # shift will take package name 'vehicle'
    # and assign it to the variable 'class'
    my $class = shift;

    my $self = {
        'distance'=> shift,
        'petrol_consumed'=> shift
    };

    # Bless function to bind the object to the
class
    bless $self, $class;

    # returning object from the constructor
    return $self;
}

# Getter-method
sub get_mileage
{
    my $self = shift;

    # Calculating0result
    my $result = $self->{'distance'} /
        $self->{'petrol_consumed'};

    print "The mileage by our vehicle is:
$result\n";
}
```

```
# The Object creation and method calling
my $obj1 = vehicle -> set_mileage(2550, 170);
$obj1->get_mileage();
```

CONSTRUCTORS AND DESTRUCTORS

We will explain constructors and destructors with examples.

Constructors

Constructors in Perl subroutines return an instance of the class as the returned object. In Perl, the constructor is usually referred to as “new.” Unlike many other OOP languages, Perl has no particular syntax for object construction. It utilizes data structures (hashes, arrays, scalars) that have been expressly connected with the class. On hash reference and the class name, the constructor calls the “bless” method (the name of the package).¹

Let’s develop some programs for better explanations.

Note: Due to the use of packages, the following code will not execute on online IDE. The following text represents a Perl class or module file. Save the file with the extension (*.pm).

```
# Declaring Package
package Area;

# Declaring Constructor method
sub new
{
    return bless {}, shift; # blessing on hashed
                           # reference (which is
empty) .
}

1;
```

The package name “Area” is stored in default array “@_” when the constructor method is called. The keyword “shift” is used to extract the package name from “@_” and pass it to the “bless” function.

```
package Area;

sub new
{
    my $class = shift; # defining shift in the $myclass
```

```

    my $self = {}; # hashed reference
    return bless $self, $class;
}
1;

```

In Perl, attributes are stored as key-value pairs in a hashed reference. Additionally, some attributes are being added to the code.

```

package Area;

sub new
{
    my $class = shift;
    my $self =
    {
        length => 3, # storing-length
        width => 4, # storing=width
    };
    return bless $self, $class;
}
1;

```

The (Area Class) code above has two attributes: length and width. Another Perl program is designed to use these attributes to gain access to them.

```

use strict;
use warnings;
use Area;
use feature qw/say/;

# creating new Area object
my $area = Area->new;

say $area->{length}; #print length
say $area->{width}; # print width

```

How to run the code:

- Save the Program with Package Area as Area.pm in a text file.
- Note: The file's name should always be the same as the package's name.

- Save the program that is used to access the attributes defined in the package as *.pl. * can be any name here (In this case, it is test.pl).
- Use the command perl test.pl to run the code saved as test.pl in the Perl command line.

Passing Dynamic Attributes

Adding Dynamic attributes to existing files:

Program: Areas.pm

```
package Area;

sub new
{
    my ($class, $args) = @_; # since values will be
                             # passed dynamically
    my $self =
    {
        length => $args->{length} || 1, # by default
value is 1 (stored)
        width => $args->{width} || 1, # by default
value is 1 (stored)
    };
    return bless $self, $class;
}

# we have added get_area function to
# calculate area as well
sub get_area
{
    my $self = shift;

    # getting area by multiplication
    my $area = $self->{length} * $self->{width};
    return $area;
}
1;
```

Program: test.pl

```
use strict;
use warnings;
```

```

use feature qw/say/;
use Area;

# pass length and width arguments
# to constructor
my $area = Area->new(
    {
        length => 2, # passing '2' as param of
the length
        width => 2, # passing '2' as param of
the width
    });

say $area->get_area;

```

Destructors

When all references to object are removed from scope, Perl automatically calls the destructor method. Destructor methods are useful if the class generates threads or temporary files that must be removed when the object is destroyed. Perl has a special method name for the destructor, “DESTROY,” which must be used when declaring the destructor.

Syntax:

```

sub DESTROY
{
    # DEFINE-Destructors
    my $self = shift;
    print "Constructor-Destroyed :P";
}

```

METHOD OVERRIDING IN OOPs

Overriding is a feature of any object-oriented computer language that allows a subclass or child class to implement a method that already exists in one of its superclasses or parents. When a subclass method has the same name, arguments or signature, and return type (or sub-type) as a method in its superclass, the subclass method is said to override the superclass method.

Method overriding signifies that the code consists of two or more methods with the same name, each of which has unique purpose and differs from the others. Thus, the literal definition of the term indicates that one

approach must take precedence over another. This idea refers to redefining a base class method in a derived class with the same method signature.

Method overriding is one manner in which Runtime Polymorphism is achieved in Perl. The version of a method that is run is dependent upon the object that invokes it. If object of the parent class is used to call the method, the version in the parent class will run; however, if an object of the subclass is used, the version in the child class will execute. In other words, the type of the object being referenced decides which version of an overridden method will be performed, not the type of the reference variable.

Method overriding in Perl is best shown with the following example.

We have a base class `vehicle` that has the methods `get_mileage()` and `get_cost()`, as well as a derived class `car` that has the methods `get_mileage()` and `get_age()`. Now, because one of the methods in both classes has the same name, their execution will follow the Method overriding concept. Let's look at the example and see how it's performed.

Base class creation:

```
# Declaration and definition of the Base class
use strict;
use warnings;

# Creating parent-class
package vehicle;

sub new
{

    # shift will take package name 'vehicle'
    # and assign it to the variable 'class'
    my $class = shift;

    my $self = {
        'distance'=> shift,
        'petrol_consumed'=> shift
    };

    # Bless function to bind object to class
    bless $self, $class;

    # returning object from the constructor
    return $self;
}
```

```

# Method for the calculating the mileage
sub get_mileage
{
    my $self = shift;

    # Calculating-result
    my $result = $self->{'distance'} /
        $self->{'petrol_consumed'};

    print "The mileage by our vehicle is: $result\n";
}

# Method for the calculating the cost
sub get_cost
{
    my $self = shift;

    # Calculating-result
    my $result = $self->{'petrol consumed'} * 70;

    print "Cost is: $result\n";
}
1;

```

Derived class creation:

```

# Declaring and defining the derived class

# Creating derived-class
package car;

use strict;
use warnings;

# Using the parent class
use parent 'vehicle';

# Overriding method
sub get_mileage
{
    my $self = shift;

```

```

# Calculating result
my $result = $self->{'distance'} /
               $self->{'petrol_consumed'};

print "The mileage by our car is: $result";
}

# Function to get age from the user
sub get_age
{
    my $self = shift;

    # Taking input from the user
    my $age = <>;

    # Printing the age
    print "The Age is: $age\n";
}
1;

```

Using objects to demonstrate the Method Overriding process:

```

# Calling objects and
# methods of each class
# using corresponding objects.

use strict;
use warnings;

# Using derived class as parent
use car;

# the object creation and initialization
my $ob1 = vehicle -> new(2550, 170);
my $ob2 = car -> new(2500, 250);

# Calling methods using the Overriding
$ob1->get_mileage();
$ob2->get_mileage();

```

As can be seen, the method from the class that is being called with the object overrides the other method with the same name but in different

classes. The “get_mileage” method on the object vehicle prints “The mileage by our vehicle is: 15” via the method declared in the class vehicle. When we execute the method in the class car “get_mileage” on the object of car, we get the output “The mileage by our car is: 10.”

Why Do We Override Methods?

As noted before, overridden methods enable Perl to offer polymorphism at runtime.

Polymorphism is crucial to OOP for the following reason: it enables a base class to describe methods that will share by all of its descendants, while subclasses may define the particular implementation of any or all of those methods. Perl also provides the “one interface, several methods” element of polymorphism through overridden methods.

Dynamic method dispatch (runtime polymorphism) is one of object-oriented architecture’s most effective strategies for code reuse and resilience. The ability to exist code libraries to call methods on instances of new classes without recompilation while preserving a clean abstract interface is a potent instrument.

Overridden methods enable us to invoke methods of any derived class without knowing the object type of the derived class. Thus, method overriding simplifies programming since there is no need to memorize distinct names when developing new methods; instead, it is more necessary to remember the processes inside the Method.

INHERITANCE IN OOPs

Inheritance is a class’ ability to extract and use the characteristics of another class. It is the process of creating new classes, known as Derived classes, from existing classes, known as Base classes. The essential idea is that the programmer may utilize the characteristics of one class in another without declaring or defining the same item several times across classes.

We may inherit the member functions from the base class instead of writing them in each class declaration. Inheritance is one of OOP’s most essential features.

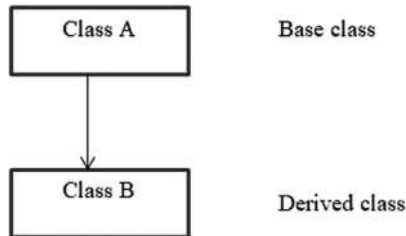
Subclass: A subclass or Derived class is a class that inherits properties from another class.

Superclass: The class whose attributes are inherited by subclasses is known as the Superclass or Base Class.

The most basic principle of inheritance is the creation or derivation of a new class from another class.

Base Class and Derived Class

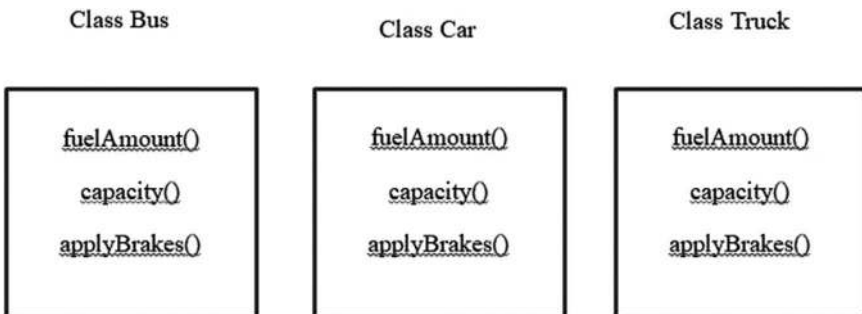
Derived classes are created by deriving additional inherited subclasses from the base class. A single base class can have multiple derived classes; this type of inheritance is known as Hierarchical Inheritance. These derived classes are all derived from a single Parent or Base class. Multiple Inheritance occurs when a derived class shares multiple parent classes and inherits its features from multiple parent classes.



Base class and Drive class.

The image above depicts the order in which a class is derived from a base class. The order and denotations shown in the above image will be used whenever necessary to theoretically depict the order of inheritance.

Consider the Vehicles class. Now we must create a class for each type of vehicle, such as a bus, car, or truck. For all Vehicles, the methods `fuelAmount()`, `capacity()`, and `applyBrakes()` will be the same.

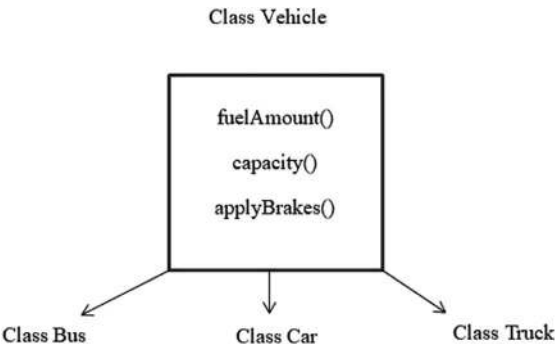


Inheritance creation.

The preceding image depicts the creation of these classes without the concept of inheritance.

The above process clearly results in the duplication of the same code three times. This raises the likelihood of error and data redundancy.

Inheritance is used to avoid situations like this. We can simply avoid data duplication and increase reusability if we create a class `Vehicle`, write these three functions in it, and inherit the rest of the classes from it. Analyze the diagram below to see how the three classes are derived from the vehicle class:

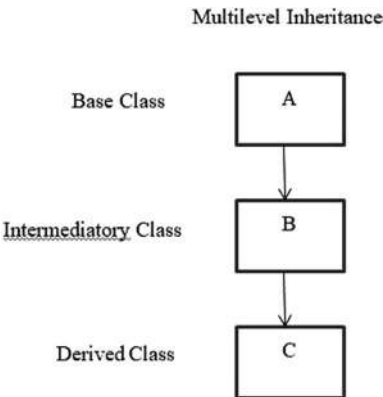


Three classes inherit from vehicle class.

Because we inherited the rest of the three classes from the base class, we only need to write the functions once instead of three times (`Vehicle`).

Multilevel Inheritance

In Perl, inheritance can take many forms, but the most common is Multilevel Inheritance, which involves a chain of base and derived classes. In Multilevel Inheritance, a derived class inherits a base class, and the derived class also serves as the base class for other classes. In the diagram below, class `A` serves as a base class for the derived class `B`, which serves as base class for the derived class `C`.



Multilevel Inheritance

Implementing Inheritance in the Perl

Packages can be used to implement inheritance in Perl. Packages are used to create a parent class from which derived classes can inherit functionality.

```
use strict;
use warnings;

# Creating parent-class
package Employee;

# Creating-constructor
sub new
{
    # shift will take package name 'employee'
    # and assign it to the variable 'class'
    my $class = shift;

    my $self = {
        'name' => shift,
        'employee_id' => shift
    };

    # Bless function to bind object to the class
    bless $self, $class;

    # returning object from the constructor
    return $self;
}
1;
```

The base class is defined in the code above. The base class is employee, and the data members are the employee id and the employee's name. This parent class code must be saved as *.pm, which we will do here as employee.pm. We'll now look at how to create a class from the previously declared base class employee.

```
# Creation of parent class
package Department;

use strict;
use warnings;
```

```
# Using the class employee as parent
use parent 'employee';

1;
```

As seen in the preceding example, the class `Department` makes use of the traits of the previously declared class `employee`. As a result, when we declared the class `Department`, we did not declare all of the data members again but instead inherited them from the base class `employee`. To run this code, save the intermediary class code as `*.pm`, as `Department.pm` in this case. This is the intermediary class and will also serve as the parent class for the following derived file `data.pl`.

```
use strict;
use warnings;

# Using the Department class as parent
use Department;

# Creating object and assigning the values
my $a = Department->new("Rajat",18017);

# Printing required fields
print "$a->{'name'}\n";
print "$a->{'employee_id'}\n";
```

Thus, inheritance is significant when working on a large project and the programmer wants to shorten the code.

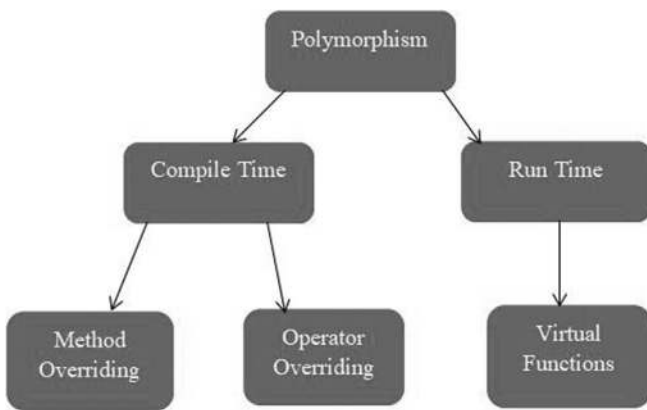
POLYMORPHISM IN OOPs

Polymorphism is the capability of processing data in several forms. Poly means numerous, and morphism implies kinds; therefore, the term itself shows the meaning. Polymorphism is one of the most fundamental concepts of an OOP language. In OOP, polymorphism is often employed when a parent class reference refers to a child class object. This section will examine how to express any function in several kinds and formats.

A real-world example of polymorphism is that a person may simultaneously play multiple roles in life. As a woman simultaneously fulfills the roles of mother, wife, employee, and daughter. Therefore, a single individual must possess various characteristics, each of which must be

implemented according to the scenario and conditions. Polymorphism is regarded as one of the OOP's most essential characteristics.

OOP's primary advantage is polymorphism. Polymorphism is so essential that languages that do not support it cannot call themselves object-oriented. Object-based languages are those that have classes but do not support polymorphism. Consequently, it is crucial for OOP language. It is the ability of an object or reference to take on multiple forms in different contexts. It supports function overloading, function overriding, and virtual functions.



Polymorphism in OOP.

Polymorphism is a property that allows any message to be sent to objects of multiple classes, and each object can respond appropriately based on the class properties.

This means that in an OOP language, polymorphism is the method that does different things depending on the class of the object that calls it. `$square->area()`, for example, will return the area of a square, whereas `$triangle->area()` may return the area of a triangle. `$object->area()`, on the other hand, would have to calculate the area based on which class `$object` was called.

Polymorphism is best explained by using the following example:

```

use warnings;

# Creation of class using package
package A;

```

```

# Constructor-creation
sub new
{

    # shift will take package name 'vehicle'
    # and assign it to the variable 'class'
    my $class = shift;
    my $self = {
        'name' => shift,
        'roll_no' => shift
    };

    sub poly_example
    {
        print("This corresponds to a class A\n");
    }
};

package B;

# The @ISA array contains a list
# of that class's parent classes, if any
my @ISA = (A);

sub poly_example
{
    print("This corresponds to a class B\n");
}

package main;

B->poly_example();
A->poly_example();

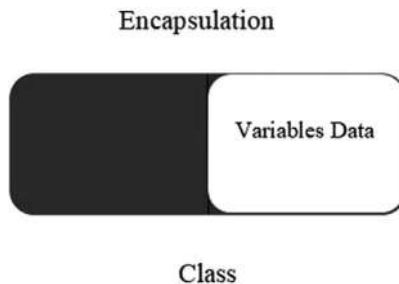
```

The method `poly_example()` defined in class B overrides the definition inherited from class A for the first output and vice versa for the second output. This allows you to add or extend the functionality of any pre-existing package without having to rewrite the entire definition of the class repeatedly. As a result, the programmer's life is made more accessible.

ENCAPSULATION IN OOPs

Encapsulation in Perl is the process of enclosing data to secure it from external sources that should not have access to that section of code. Encapsulation is a component of OOP; it ties data to the subroutines that change it. Encapsulation is, in another sense, a protective barrier that prevents the data from being accessible by code outside of this shield.

- Technically, under encapsulation, variables or data of a class are concealed from other classes and may only be accessible through member functions of the class in which they are defined.
- As with encapsulation, the data inside a class is concealed from other classes; hence, it is sometimes referred to as data-hiding.
- Encapsulation may be performed by declaring all class variables as local and retrieving class methods by importing packages to set and retrieve variable values.



Encapsulation in OOP.

Consider a real-world example of the encapsulation: in a business, there are several departments, such as accounts, finance, and sales. The finance sector conducts all financial transactions and maintains records of all financial data.

Similarly, the sales department conducts all sales-related operations and maintains sales data. Now, there may be a situation in which a finance department official needs complete sales information for a given month. In this instance, he is not permitted direct access to the sales section's data. He must first call another officer in the sales department and then request that individual data be provided.

This describes encapsulation. Here, the sales section's data and the people who may change them are grouped under the name "sales section."

Example:

```
# Declaration and definition of the Base class
use strict;
use warnings;

package Student;
sub new
{

    # shift will take package name 'Student'
    # and assign it to the variable 'class'
    my $class = shift;

    my $self = {
        'name'=> shift,
        'age'=> shift,
        'roll_no' => shift
    };

    # Bless function to bind the object to class
    bless $self, $class;

    # returning object from the constructor
    return $self;
}

# Method for the displaying the details
sub get_details
{
    my $self = shift;

    print "The Name is: $self->{'name'}\n";
    print "The Age is: $self->{'age'}\n";
    print "The Roll_no is: $self->{'roll_no'}";
}

# The Object creation and calling
my $obj1 = Student->new("Rohini", 27, 15);
$obj1->get_details();
```

If there is a need to access the class' data for any modifications or simply to print the class' data in the above code, it cannot be done directly. It is

essential to create an object of that class and then use the `get_details()` method to access the data. This is referred to as Data Encapsulation.

The benefits of encapsulation are:

- **Data-hiding:** The user will be unsure of the inner implementation of the class. The user cannot see how the class stores data in its variables. They are just conscious that we are giving values to accessors, and variables are being initialized with those values.
- **Enhanced flexibility:** We may make the class variables read-only or write-only depending on our needs. To make variables read-only, we need to utilize Get Accessor in the code. To make variables read-only, we must only utilize the Set Accessor.
- **Reusability:** Encapsulation also promotes reusability and makes it simple to adapt to new needs.
- **Verifying code is simple:** Encapsulated code is simple to unit test.

This chapter covered OOP in Perl with classes and objects, methods, constructors, and destructors. Furthermore, we discussed method overriding, inheritance, polymorphism, and encapsulation.

NOTE

-
1. Perl | Constructors and Destructors.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Subroutines in Perl

IN THIS CHAPTER

- Function Signature
- Passing Complex Parameters to a Subroutine
- Mutable and Immutable parameters
- Multiple Subroutines
- Use of `return()` Function
- Pass By Reference
- Recursion

In the previous chapter, we discussed object-oriented programming in Perl, and in this chapter, we will cover subroutines.

SUBROUTINES OR FUNCTIONS

A Perl function or subroutine is a collection of statements that execute a specific purpose. In any programming language, code reuse is desired. Therefore, the user places the chunk of code in a function or subroutine to eliminate the need to write code repeatedly. In Perl, the phrases function, subroutine, and method are synonymous, although they are distinct in other programming languages. The term subroutines is the most common in Perl programming since it is formed with the keyword `sub`. Whenever a function is called, Perl suspends the execution of the whole program,

jumps to the function to execute it, and then returns to the previous area of code. One need not use the return statement.

Determining Subroutines

The general form of subroutine definition in Perl is as follows:

```
sub subroutinename
{
    # body of the method or subroutine
}
```

Calling Subroutines

Subroutines in Perl can invoke by passing the arguments list to it as follows:

```
subroutineName (argumentslist);
```

The method described above will only work with Perl versions 5.0 and later. There was another way to call the subroutine before Perl 5.0, but it is not recommended because it bypasses the subroutine prototypes.

```
&subroutineName (argumentslist);
```

Example:

```
# Program to demonstrate
# the subroutine declaration and calling

#!/usr/bin/perl

# defining-subroutine
sub ask_user {
    print "Hello Everyone\n";
}

# calling-subroutine
# we can also use
# &ask_user();
ask_user();
```

Passing Parameters to Subroutines

This is used to pass values as arguments to subroutines. This is accomplished using the special list array variables “\$_.” This will be assigned to the functions as \$_[0], \$_[1], etc.

Example:

```
# Program to demonstrate
# the Passing parameters to subroutines

#!/usr/bin/perl

# defining-subroutine
sub area
{
    # passing-argument
    $side = $_[0];

    return ($side * $side);
}

# calling-function
$totalArea = area(4);

# displaying-result
printf $totalArea;
```

Subroutines have the following advantages:

- It allows us to reuse code and makes error detection and debugging easier.
- It aids in the structural organization of the code.
- Sections of code are organized in chunks.
- It improves code readability.

A Perl function or subroutine is a collection of statements that perform a specific task. Every programming language user wishes to reuse code. So the user puts the code section in a function or subroutine to avoid having to write the code repeatedly. We will go over the following ideas:

- Hashes are passed to subroutines.
- Lists are passed to subroutines.
- Returning value from subroutine local and global variables.
- Variable number of parameters in a subroutine call.

Passing Hashes to Subroutines

A hash can also be passed to subroutines, which convert it to its key-value pair.

Example:

```
# Perl program to demonstrate the
# passing of hash to subroutines

#!/usr/bin/perl

# Subroutine-definition
sub Display_hash {

# hash variable to store the
# passed arguments
my (%hash_var) = @_;

# to display passed list elements
foreach my $key (keys %hash_var )
{
    my $val = $hash_var{$key};
    print "$key : $val\n";
}
}

# defining-hash
%hash_para = ('Subject' => 'Perl Programing',
'Marks' => 69);

# calling Subroutine with the hash parameter
Display_hash(%hash_para);
```

Passing Lists to Subroutines

Because `@_` is a special array variable within a function or subroutine, it is used to pass lists to the subroutine. Perl accepts and parses arrays and lists differently, making it difficult to extract the discrete element from `@_`. To pass a list along with the other scalar arguments, the list must be passed as the last argument.

Example:

```
# program to demonstrate
# passing of lists to subroutines

#!/usr/bin/perl

# Subroutine-definition
sub Display_List {

# array variable to store the
# passed arguments
my @para_list = @_;

# to print passed list elements
print "Given list is @para_list\n";
}

# passing the scalar argument
$sc = 100;

# passing-list
@li = (20, 30, 40, 50);

# Calling Subroutine with the scalar
# and list parameter
Display_List($sc, @li);
```

Returning a Value from a Subroutine

A subroutine, like any other programming language, can return a value. If the user does not manually return a value from the subroutine, the subroutine will return a value automatically. In this case, the automatically returned value will result from the last calculation performed in the subroutine. The result can be a scalar, an array, or a hash.

Example:

```
# Program to demonstrate
# the returning values subroutine

#!/usr/bin/perl

# subroutine-definition
sub Sum {
```

```

# To get total number
# of the parameters passed.
$num = scalar(@_);
$s = 0;

foreach $i (@_)
{
    $s += $i;
}

# returning-sum
return $s;
}

# subroutine calling and storing the result
$result = Sum(40, 4, 50);

# displaying the result
print "Sum of given numbers : $result\n";

```

Local and Global Variables in Subroutines

By default, all variables within a Perl program are global variables. However, using my keyword, you can create local or private variables within a block. A private variable has a limited scope, such as between blocks (if, while, for, foreach, and so on) and methods. These variables cannot be used outside of a block or method.

Example:

```

# program to demonstrate the Local
# and Global variables in the subroutine

#!/usr/bin/perl

# Global variable
$str = "PeeksforPeeks";

# subroutine-definition
sub Peek {

# Private variable by using my
# keyword for Peek function
my $str;

```

```

$str = "PFP";
print "Inside Subroutine: $str\n";
}

# Calling-Subroutine
Geeks();

print "Outside Subroutine: $str\n";

```

A Varying Number of Parameters in a Subroutine Call

Perl does not include any built-in facilities for declaring the parameters of a subroutine, making it very simple to pass any number of parameters to a function.

Example:

```

# program to demonstrate the variable
# number of parameters to the subroutine

#!/usr/bin/perl

use strict;
use warnings;

# defining-subroutine
sub Multiplication {

    # private variable containing
    # the default value as 1
    my $mul = 1;

    foreach my $val (@_)
    {
        $mul *= $val;
    }

    return $mul;
}

```

```
# Calling the subroutine with 4 parameters
print Multiplication(9, 4, 2, 5);

print "\n";

# Calling subroutine again but
# with the 3 parameters
print Multiplication(4, 6, 3);
```

In general, passing more than one array or hash as a parameter to a subroutine causes it to lose its distinct identity. Likewise, returning more than one array or hash from a subroutine results in the loss of their distinct identities. We can solve these issues by making use of references.

FUNCTION SIGNATURE IN PERL

A Perl function or subroutine is a collection of statements that perform a specific task. Every programming language user wishes to reuse code. So the user places the code section in a function or subroutine to avoid having to write the code repeatedly. Although the terms function, subroutine, and method are interchangeable in Perl, they are not in other programming languages. Because it is created with the keyword `sub`, the terms subroutines are commonly used in Perl programming. When a function is called, Perl stops executing its entire program and jumps to the function to execute it before returning to the section of code that was previously running. The return statement can be avoided.

Defining Subroutines

In Perl, the general form of defining a subroutine is as follows:

```
sub subroutinename
{
    # body of the method or subroutine
}
```

Function Signature

When a function is constructed, a set of parameters is also provided inside the parenthesis to specify the sort of arguments the function will receive on its call. This function signature may include one or several arguments.

A subroutine or function with a defined signature can only accept arguments of the type specified in its signature. An error will occur if the inputs given to the procedure do not match its signature. A function signature reveals a great deal about the subroutine type. It allows the user to create subroutines with the same name but distinct signatures, i.e. different arguments. Different subroutines may have the same name in Perl, but their arguments must be distinct.

Example:

```
sub examplefunc($variable)
{
    statement;
}

sub examplefunc($variable1, $variable2)
{
    statement;
}
```

In the above code, the names of the subroutines are identical, but their parameter counts are distinct; as a result, they are not deemed to be distinct subroutines, and our Perl code will not raise an error.

Passing Parameters of a Type other than that Specified in the Signature

When a subroutine is defined with a signature, it must take arguments of the same type as its signature. If an argument different than the function signature is given, an error will be generated, causing the code compilation to fail.

Example:

```
#!/usr/bin/perl

# Defining the Function Signature
sub example(Int $variable)
{
    return $variable / 2;
}

# Function-Call
print example(66);
```

If we pass an argument of a type other than the function signature, the following error will be generated:

```
#!/usr/bin/perl

# Defining the Function Signature
sub example(Int $variable)
{
    return $variable / 2;
}

# Function Call with the
# string type parameter
print example("66");
```

Difference in Number of Arguments

When a function signature is defined, it also includes the number and type of arguments that can be passed to it. If we call the function with a different number of arguments, Perl will throw an error because functions with different signatures have different meanings.

Example:

```
#!/usr/bin/perl

# Defining Function-Signature
sub example(Int $variable)
{
    return $variable / 2;
}

# Function Call with the
# two arguments
print example(66, 49);
```

Function signature is a useful feature, but it can be very annoying for some programmers because it restricts the use of the function to a specific type of parameter. There are no restrictions on the type of parameters that can pass to the function if it is defined without a signature.

Example:

```
#!/usr/bin/perl

# Defining Function-Signature
sub example($variable)
{
    return $variable / 2;
}

# Function Call with the Integer argument
# written as string type
print example("66");
```

The above code declares a function with no specific argument type, so when an integer value is passed to it as a string, it automatically converts the argument to integer form and returns the result. However, if we call the function with a string argument and the operation to be performed on it requires an integer value, we will get the following error:

```
#!/usr/bin/perl

# Defining Function-Signature
sub example($variable)
{
    return $variable / 2;
}

# Function Call with the
# string type argument
print example("Peeks");
```

PASSING COMPLEX PARAMETERS TO A SUBROUTINE

A Perl function or subroutine is a collection of statements that perform a specific task. The user wants to reuse code in every programming language. So the user places the code section in a function or subroutine to avoid having to rewrite the same code repeatedly. As a result, function or subroutine is used in all programming languages. These functions or subroutines can accept various data structures as parameters. Some of these are discussed further below:

- Passing lists or arrays to a subroutine
- Passing references to a subroutine

- Passing file handles to a subroutine
- Passing hashes to a subroutine

Passing arrays or lists to a subroutine: An array or list can be passed as a parameter to the subroutine, and an array variable `@_` is used to accept the list value inside the subroutine or function.

First example: A single list is passed to the subroutine, and its elements are displayed.

```
#!/usr/bin/perl

# Defining-Subroutine
sub Display_List
{

    # array variable to store the
    # passed arguments
    my @List1 = @_;

    # Printing passed list elements
    print "Given list is @List1\n";
}

# Driver-Code

# passing-list
@list = (1, 2, 3, 4);

# Calling the Subroutine with
# list-parameter
Display_List(@list);
```

Second example: In this case, two lists are passed to the subroutine, and their contents are displayed.

```
#!/usr/bin/perl

# Defining-Subroutine
sub Display_List
```

```

{

    # array variable to store
    # the passed arguments
    my @List3 = @_;

    # Printing the passed lists' elements
    print "The Given lists' elements are @List3\n";
}

# Driver-Code

# passing lists
@List1 = (11, 12, 13, 14);
@List2 = (20, 30, 40, 50);

# Calling Subroutine with
# the list parameters
Display_List(@List1, @List2);

```

Third example: A scalar argument and a list are passed to the subroutine, and the list elements are displayed.

```

#!/usr/bin/perl

# Defining-Subroutine
sub Display_List
{

    # array variable to store
    # passed arguments
    my @List2 = @_;

    # Printing passed list and scalar elements
    print "The List and scalar elements are @List2\n";
}

# Driver-Code

# passing lists
@List = (11, 12, 13, 14);

```

```
# passing the scalar argument
$scalar = 100;

# Calling Subroutine with
# the list parameters
Display_List(@List, $scalar);
```

Passing references to subroutines: References can also be passed as parameters to subroutines. The array reference is passed to the subroutine, and the maximum value of the array elements is returned.

Example:

```
#!/usr/bin/perl
use warnings;
use strict;

# Creating an array of the some elements
my @Array = (20, 30, 40, 50, 60);

# Making reference to the above array
# and calling the subroutine with
# reference of the array as the parameter
my $m = max(\@Array);

# Defining-subroutine
sub max
{

    # Getting array elements
    my $Array_ref = $_[0];
    my $k = $Array_ref->[0];

    # Iterating over each element of
    # the array and finding maximum value
    for(@$Array_ref)
    {
        $k = $_ if($k < $_);
    }
    print "Max of @Array is $k\n";
}
```

Passing a hash to a subroutine: A hash can also be passed as a parameter to a subroutine, which displays its key-value pair.

Example:

```
#!/usr/bin/perl

# Subroutine-definition
sub Display_Hash
{

    # Hash variable to store the
    # passed arguments
    my (%Hash_var) = @_;

    # Displaying passed list elements
    foreach my $key (keys %Hash_var)
    {
        my $value = $Hash_var{$key};
        print "$key : $value\n";
    }
}

# Driver-Code

# defining hash
%Hash = ('Company' => 'PeeksforPeeks',
         'Location' => 'Delhi');

# calling Subroutine with the hash parameter
Display_Hash(%Hash);
```

Passing file handles to a subroutine: To create a file or access its contents, a FileHandle is required. A FileHandle is nothing more than a structure used with the operators to access the file in a specific mode, such as reading, writing, appending, and so on. FileHandles can also be passed as a parameter to a subroutine to perform various operations on Files.

A FileHandle is passed to a subroutine in the following example:

Example:

```
#!/usr/bin/perl

sub printem
```

```

{
    my $file = shift;

    while (<$file>) {
        print;
    }
}

open(fh, "Hello1.txt") or die "File '$filename'
cannot be opened";

printem *fh;

```

MUTABLE AND IMMUTABLE PARAMETERS

A Perl function or subroutine is a collection of statements that perform a specific task. The user wants to reuse code in every programming language. So the user places the code section in a function or subroutine to avoid having to write the code repeatedly. These subroutines have parameters that define the type and number of arguments that can be passed to the function when it is called. These arguments are used to evaluate the values passed to the function using the function's expressions. These values are then passed to the calling function and returned to the specified parameters.

Example:

```

sub Function1(parameter-1, parameter-2)
{ statement; }

```

Mutable Parameters

These are parameters, the values of which are changed inside the function to which they have been passed.

It indicates that when a parameter is supplied to a function through the caller function, its value is tied to the parameter in the called function, thus any changes made to the value in the called function will also be reflected in the parameter passed by the caller function.

Immutable Parameters

These arguments are of those kinds, the values of which cannot be altered inside the function to which they are supplied. It indicates that the

subroutine gets a value rather than a variable when a parameter is supplied to the function through the caller function. Consequently, any modifications to the function parameter are not reflected.

Perl's subroutine parameters are immutable by default, meaning they cannot modify inside a function, and one cannot inadvertently alter the arguments of the calling function. In specific programming languages, this is referred to as “call by value,” which indicates that the subroutine being called gets a value rather than the variable, so the parameters of the calling function are not altered.

Example:

```
#!/usr/bin/perl

# Function-Definition
sub Func(Int $variable)
{
    # Operation to perform
    $variable /= 2;
}

# Defining local variable
my $value = 40;

# Function Call with the local variable
print Func($value);
```

Traits are used to change the properties of these parameters. The value of a parameter can be changed within a subroutine using traits.

Traits

These are predefined built-in subroutines that alter the method's behavior when run at compile time when used within a method. Traits can even be used to modify the method's body or tag the method with metadata.

Depending on their usage, traits can be of various types, such as:

- is cached trait automatically caches the function's return value based on the arguments passed to it.
- is rw trait enables writable accessors to subroutine parameters.

- is copy trait allows us to change the value of a parameter within a subroutine but not the argument in the caller function.

Example: Use of the is copy trait

```
#!/usr/bin/perl

# Function Definition using the
# 'is copy' trait
sub Func(Int $variable is copy)
{
    # Operation to be performed
    $variable += 5;
}

# Defining the local variable
my $value = 20;

# Function Call with the local variable
print Func($value), "\n";

# Checking if
# the $value gets updated or not
print $value;
```

Because Perl does not allow the modification of arguments within a subroutine by default, the copy trait is used in the above code. This trait enables the program to assign the caller function's parameter value to the subroutine being called a parameter. This trait, however, will only affect the value of the argument in the called function.

Example: Use of the is rw trait

```
#!/usr/bin/perl

# Function Definition using the
# 'is rw' trait
sub Func(Int $variable is rw)
{
    # Operation to perform
    $variable += 5;
}
```

```
# Defining local variable
my $value = 20;

# Function Call with the local variable
print Func($value), "\n";

# Checking if
# the $value gets updated or not
print $value;
```

When `is rw` is used instead of `is copy` trait in the preceding code, the value of argument passed in the caller function is also updated.

When the `is rw` trait is used, the argument of the called function is bound with the argument of the caller function, so if the former value changes, the latter is immediately updated. This is because a process known as “call by reference.” Because both arguments refer to the same memory location (because of the `is rw` trait); as a result, the parameters are fully mutable.

MULTIPLE SUBROUTINES

A Perl function or subroutine is a collection of statements that perform a specific task. Every programming language user wishes to reuse code. So the user places the code section in a function or subroutine to avoid having to write the code repeatedly.

Although the terms function, subroutine, and method are interchangeable in Perl, they are not in other programming languages. Because it is created with the keyword `sub`, the terms subroutines are commonly used in Perl programming. When a function is called, Perl stops executing its entire program and jumps to the function to execute it before returning to the section of code that was previously running. The `return` statement can be avoided.

Subroutine Definition

In Perl, the general form of defining a subroutine is as follows:

```
sub subroutine_name
{
    # body of the method or subroutine
}
```

Because Perl allows you to write multiple subroutines with the same name unless they have different signatures, a program in Perl can contain

multiple subroutines with the same name without generating an error. This can be defined by varying the arity of each subroutine with the same name.

Subroutine arity: Perl subroutines can have the same name as long as they have a different set of arity. The number of arguments a subroutine has is referred to as its arity. These arguments could be of a different data type or not. The Perl program will not generate an error as long as the number of subroutines differs.

Use of the “multi” Keyword

The keyword “multi” can use to create multiple subroutines in Perl. This is useful for creating multiple subroutines with the same name.

Example:

```
multi Func1($var){statement};
multi Func1($var1, $var2){statement-1;
statement-2;}
```

Multiple subroutines are commonly used to create built-in functions and most operators in a programming language such as Perl. The program’s complexity is reduced by not using different names for each other subroutine. Whatever code statement is required, simply pass the number of arguments for that function, and the work will be completed. In this case, compiler will select the version of the subroutine, the function signature of which matches the one invoked.

Various programs, such as factorial of a number, Fibonacci series, and so on, require the use of more than one function to solve the problem. The use of multiple subroutines will aid in the reduction of the complexity of such programs.

First example: Fibonacci series sum

```
#!/usr/bin/perl
# Program to print sum of the fibonacci series

# Function for the $n = 0
multi Fibonacci_func(0)
{
    1; # returning 1
}
```

```
# Function for the $n = 1
multi Fibonacci_func(1)
{
    1; # returning 1
}

# Recursive function to calculate the Sum
multi Fibonacci_func(Int $n where $n > 1)
{
    Fibonacci_func($n - 2) +
    Fibonacci_func($n - 1);
}

# Printing sum
# using the Function Call
print Fibonacci_func(27);
```

To calculate the sum of Fibonacci series, the preceding example employs multiple subroutines.

Second example: A number factorial

```
#!/usr/bin/perl
# Program to print factorial of number

# Factorial of 0
multi Factorial(0)
{
    1; # returning 1
}

# Recursive Function
# to calculate the Factorial
multi Factorial(Int $n where $n > 0)
{
    $n * Factorial($n - 1); # Recursive-Call
}

# Printing result
# using the Function Call
print Factorial(25);
```

In the preceding examples, the program declares multiple subroutines with the same name but different arity using the “multi” keyword.

return() FUNCTION

In Perl, the `return()` function returns value at the end of a subroutine, block, or `do` function. The returned value could be a scalar, an array, or a hash depending on the context.

Syntax: `return Value`

Returns:

a List in the Scalar Context

If no argument is passed to the `return` function, it returns an empty list in the list context, `undef` in the scalar context, and nothing in the void context.

First example:

```
#!/usr/bin/perl -w

# Subroutine for the Multiplication
sub Mul($$)
{
    my($x, $y) = @_;
    my $z = $x * $y;

    # Return-Value
    return($x, $y, $z);
}

# Calling in Scalar context
$retval = Mul(35, 20);
print ("The Return value is $retval\n" );

# Calling in the list context
@retval = Mul(35, 20);
print ("The Return value is @retval\n" );
```

Second example:

```
#!/usr/bin/perl -w

# Subroutine for the Subtraction
sub Sub($$)
```

```

{
    my($x, $y ) = @_;

    my $z = $x - $y;

    # Return Value
    return($x, $y, $z);
}

# Calling in the Scalar context
$retval = Sub(35, 20);
print ("The Return value is $retval\n" );

# Calling in the list context
@retval = Sub(35, 20);
print ("The Return value is @retval\n" );

```

REFERENCES IN PERL

Variables are used in Perl to access data stored in memory (all data and functions are stored in memory). Variables are given data values that are then used in various operations. Perl reference allows you to access the same data but through a different variable. In Perl, a reference is a scalar data type that holds the location of another variable. Other variables include scalars, hashes, arrays, function names, and so on. Nested data structures are simple to construct because a user can create a list that contains references to another list that contains references to arrays, scalar, hashes, and so on.

Making a Reference

We can make references to scalar value, hash, array, function, and so on. To make a reference, define a new scalar variable and prefix the variable's name (the reference of which we want to make) with a backslash.

Making references to different data types:

```

# Array-Reference

# defining-array
@array = ('1', '2', '3');

# making reference of the array variable
$reference_array = \@array;

```

```
# Hash-Reference

# defining-hash
%hash = ('1'=>'a', '2'=>'b', '3'=>'c');

# make reference of hash variable
$reference_hash = \%hash;

# Scalar Value Reference

# defining-scalar
$scalar_val = 2431;

# making reference of the scalar variable
$reference_scalar = \$scalar_val;
```

Notes:

- The curly brackets {} around key and value pairs can be used to create a reference to an anonymous hash.

Example:

```
# creating reference to the anonymous hash
$ref_to_anonymous_hash = {'PFP' => '1', 'Peeks'
=> '2'};
```

- The square brackets [] can use to create a reference to an anonymous array.

Example:

```
# creating reference to anonymous array
$ref_to_anonymous_array = [30, 40, ['P', 'F',
'P']];
```

- Sub can also be used to create a reference to an anonymous subroutine. There will be no name for the sub here.

Example:

```
# creating reference to anonymous subroutine
$ref_to_anonymous_subroutine = sub { print
"PeeksforPeeks\n"};
```

- It is not possible to create reference to an input/output handle, such as dirhandle or FileHandle.

Dereferencing

Now that we've made the reference, we must use it to get to the value. Dereferencing is a method of gaining access to the value in the memory pointed to by the reference. Depending on the type of variable, we use prefix \$, @, percent, or & to dereference (reference can point to a array, scalar, hash, etc.).

First example:

```
# Program to illustrate
# Dereferencing of an Array

# defining an array
@array = ('1', '2', '3');

# making reference to an array variable
$reference_array = \@array;

# Dereferencing
# printing value stored
# at $reference_array by prefixing
# @ as it is array reference
print @$reference_array;
```

Second example:

```
# program to illustrate
# the Dereferencing of a Hash

# defining hash
%hash = ('1'=>'a', '2'=>'b', '3'=>'c');

# creating reference to hash variable
$reference_hash = \%hash;

# Dereferencing
# printing value stored
# at $reference_hash by prefixing
# % as it is a hash reference
print %$reference_hash;
```

Third example:

```
# program to illustrate
# Dereferencing of Scalar

# defining scalar
$scalar = 2431;

# creating reference to scalar variable
$reference_scalar = \$scalar;

# Dereferencing
# printing value stored
# at $reference_scalar by prefixing
# $ as it is Scalar reference
print $$reference_scalar;
```

PASS BY REFERENCE

When variable is passed by reference, the function operates on the function's original data. The function can change the original value of a variable by passing it by reference.

When the values of the elements in the argument arrays @_ are changed, so do the values of the corresponding arguments. This is what parameter passing by reference accomplishes.

In the following example, updating the elements of an array @a in subroutine sample changes the value of the parameters, which is reflected throughout the program. As a result, when parameters are referenced, changing their value in the function changes their value in the main program.

First example:

```
#!/usr/bin/perl

# Initialising array 'a'
@a = (0..10);

# Array before the subroutine call
print("The Values of an array before function
call: = @a\n");

# calling subroutine 'sample'
sample(@a);
```

```
# Array after the subroutine call
print("The Values of an array after function call:
= @a");

# Subroutine to represent
# Passing by Reference
sub sample
{
    $_[0] = "A";

    $_[1] = "B";
}
```

The program operates as follows:

- An array with the values ranging from 0 to 10 is defined.
- This array is then passed to the “sample” subroutine.
- A subroutine called “sample” has already been defined. The first and second parameters’ values are changed within this via the argument array @_.
- After calling the subroutine, the values of the array @a are displayed. The first two scalar values are now updated to A and B, respectively.

Second example:

```
#!/usr/bin/perl

# Initializing values to scalar
# variables x and y
my $x = 10;
my $y = 20;

# Values before the subroutine call
print "Before calling the subroutine x = $x,
y = $y \n";

# Subroutine call
sample($x, $y);

# Values after the subroutine call
print "After calling the subroutine x = $x, y = $y ";
```

```
# Subroutine sample
sub sample
{
    $_[0] = 1;
    $_[1] = 2;
}
```

PERL RECURSION

Recursion is a mechanism that allows a function to call itself repeatedly until the required condition is met. When the function call statement is written within the same function, the function is said to be recursive.

The argument passed to a function is retrieved from default array `@_`, with each value accessible via `$_[0]`, `$_[1]`, and so on.

First example: The following example computes the factorial of a number.

The Factorial of any number n is
 $(n) * (n-1) * (n-2) * \dots * 1$.

e.g.:

$4! = 4 * 3 * 2 * 1 = 24$

$3! = 3 * 2 * 1 = 6$

$2! = 2 * 1 = 2$

$1! = 1$

$0! = 0$

```
#!/usr/bin/perl
```

```
# Program to calculate Factorial
```

```
sub fact
```

```
{
```

```
# Retrieving first argument
```

```
# passed with the function-calling
```

```
my $x = $_[0];
```

```
# checking if value is 0 or 1
```

```
if ($x == 0 || $x == 1)
```

```
{
```

```
    return 1;
```

```
}
```

```
# Recursively calling function with next value
```

```
# which is one less than current one
```

```

else
{
    return $x * fact($x - 1);
}
}

# Driver-Code
$a = 5;

# Function call and printing the result after return
print "The Factorial of a number $a is ", fact($a);

```

The program works as follows:

- Step 1: When the value of scalar *a* is 0 or 1, the function returns 1 because both 0! and 1! are 1.
- Step 2: If the value of the scalar *a* is 2, then `fac(x-1)` calls `fac(1)`, which returns 1.
 As a result, $2 * \text{factorial}(1) = 2 * 1 = 2$.
 As a result, it will return 2.
- Step 3: Similarly, when higher values are passed to the function, the argument value decreases by 1 with each call and computes until the value reaches 1.

Second example: The following example computes the Fibonacci series up to a given number.

```

#!/usr/bin/perl

# Program to print Fibonacci series
sub fib
{
    # Retrieving values from parameter
    my $c = shift;
    my $d = shift;

    # Number till which the series is to print
    my $n = shift;

    # Check for end value
    if ($d > $n)

```

```

    {
        return 1;
    }

    # Printing number
    print " $d";

    # Recursive Function Call
    fib($d, $c + $d, $n);
}

# Driver Code

# Number till which series is to print
$m = 5;

# First two elements of series
$x = 0;
$y = 1;

print "$x";

# Function call with the required parameters
fib($x, $y, $m);

```

Here's how the program works:

- Step 1: The function `fib()` is called with 3 parameters, the first two of which are 0 and 1, while `$n` is number till which series is to be printed.
- Step 2: These values are transferred in the form of an array, the contents of which are retrieved using `shift`.
- Step 3: For each call, the first two values are retrieved using `shift` and stored in scalars `c` and `d`. These two values are now added to yield the next value in the series. This step is repeated until the value reaches the end value specified by the user.

In this chapter, we covered subroutines in Perl where we discussed function signature, passing complex parameters to a subroutine, and mutable and immutable parameters. Moreover, we discuss multiple subroutines use of `return()` function, pass by reference, and recursion.

Appraisal

Perl is a widely used cross-platform, open-source programming language in both the commercial and private computing industries. Perl was favored by Web developers in the late 20th and early 21st centuries because of its adaptable text-processing and problem-solving skills.

Larry Wall, an American programmer and linguist, published Perl 1.0 for Unix-based computers for the first time in December 1987. This early version of Perl was an intuitive, easily coded language for reading, extracting, and printing information from text files; it was also capable of doing other systems management tasks. Perl, which is frequently believed to stand for “practical extraction and report language,” was inspired by existing programming languages, such as C, BASIC, and AWK, but its wide usage of common English terminology reflected Wall’s linguistic background. Perl was a milestone product in promoting the open-source model, a collaborative approach to software development as opposed to a proprietary one.

Perl is a programming language designed for script manipulation. However, Perl is now used for a wide range of purposes, including web development, graphical user interface (GUI) development, system administration, and many others. It is a reliable and cross-platform programming language.

Perl CGI is used for web development. CGI is the system gateway that communicates with the web browser and Perl.

Its most common application is extracting information from a text file and printing a report to convert a text file into another format. This is due to its name being derived from the phrase “Practical Extraction and Report Language.”

Perl scripts are programs written in Perl, whereas Perl programs are system programs that execute Perl scripts. Perl is a scripting language. When we run a Perl program, it is first compiled into byte code and then converted into machine instructions. Writing something in Perl rather than C saves time.

It works with the vast majority of operating systems and is included in the Oxford English Dictionary. It borrows concepts and syntax from a variety of languages, including AWK, bourne shell, C, sed, and even English.

HISTORY OF PERL

Larry Wall created Perl in 1987 as a scripting language to help with report processing.

On December 18, 1987, it was released in version 1.0.

Perl 2 was released in 1988, and it included a much improved regular expression engine.

Perl 3 was released in 1989, and it included support for binary data streams.

Perl 4, released in 1991, had better documentation than previous versions.

On October 17, 1994, Perl 5 was released. Its most recent version included many new features, such as objects, variables, references, and modules.

The most recent version, 5.24, was released on May 9, 2016.

FEATURES OF PERL

It has a straightforward object-oriented programming syntax.

It can easily expand because it supports 25,000 open-source modules.

Unicode is supported.

It includes powerful tools for processing text to make it compatible with mark-up languages, such as HTML and XML.

It supports third-party databases, such as Oracle and MySQL.

It can embed in other systems like web servers and database servers.

It is GNU-licensed open-source software.

Perl is used to write many frameworks.

It is capable of processing encrypted web data, including e-commerce transactions.

It is a platform-independent language.

It includes a regular expression engine that can transform any type of text.

LICENSING FOR PERL

Larry Wall owns the copyright (C) to Perl 5. It is free and open-source software. It may be redistributed or modified under the terms of the GNU and Artistic Licenses.

The GNU General Public License offers its users free and open-source software. Any program derived from GNU-licensed source code must use the same license.

According to the Artistic license, a package derived from Perl must clearly highlight its modifications. The original module, as well as the derived one, should be distributed. Above all, the original author must be acknowledged as the package's owner. Users should be able to distinguish between original and derived modules.

PERL AND THE WEB

Because of its text manipulation capabilities and quick development cycle, Perl was once the most popular web programming language.

Perl is commonly referred to as “the duct tape of the Internet.”

Perl is capable of processing encrypted Web data, including e-commerce transactions.

Perl can be embedded in web servers to speed up processing by up to 2000%.

The Perl module `mod Perl` enables the Apache web server to include a Perl interpreter.

Perl's DBI package simplifies web-database integration.

PERL IS INTERPRETED

Because Perl is an interpreted language, our code can run without being compiled into a non-portable executable program.

Traditional compilers translate programs into machine code. When you run a Perl program, it is first compiled into byte code, which is then converted into machine instructions (as the program runs). As a result, it differs from shells or Tcl, which are strictly interpreted without an intermediate representation.

It's also unlike most C or C++ versions, which are compiled directly into machine-dependent formats. It is in the middle, alongside Python, AWK, and Emacs. `elc` files.

ADVANTAGES

Perl has the following advantages:

- **Options:** Perl users have numerous options for writing programs or solving problems.

- **Flexible:** The language's design and syntax allow users to code in their preferred programming style.
- **Open source:** It's free to use. Anyone can access, develop, and use Perl on various platforms for free.
- **Availability:** It comes pre-installed in many places, and the Comprehensive Perl Archive Network contains over 25,000 Perl modules. The majority of operating systems also support it.

DISADVANTAGES

The main disadvantage of Perl is that it is a relatively messy language in a variety of ways:

- **Hard to read:** It's difficult to read. According to some developers, Perl is more difficult to read and less streamlined than newer languages such as Python. Because of the numerous ways to write a Perl program, the code can become disorganized and untidy.
- **Difficult to debug:** Debugging and fixing problems can be difficult because Perl code can be obscure or messy.
- **Flaws in performance:** The same flexibility that makes Perl useful can also make it slow. This is because flexibility can lead to inefficiencies and redundancy, making compilation time longer.

PERL VS PYTHON

Perl and Python have a common ancestor. Both were created to make scripting easier. Perl was created to give Unix scripts a C-like structure. Python was created to make C easier to use and ready for scripting.

Syntactically, Perl and Python are very similar, and translating from Perl to Python is relatively simple with only a few significant syntax changes.

However, there are four significant differences:

- A semicolon terminates lines in Perl.
- Python lacks curly braces and indentation, whereas Perl does.
- Variable names in Perl are styled with variables, such as \$x, % x, and x. In Python, variable names are styled without a variable indicator, such as x.
- In Python, the print statement inserts a new line at the end of the output.

Other differences include:

- Perl can be messy, whereas Python is more streamlined.
- In Perl, numerous ways exist to accomplish a task, whereas Python is designed to provide a single clear path to any given function.
- Python is more readable than Perl.
- Python is commonly considered one of the best programming languages to learn and one of the most user-friendly for new developers.
- Python is a newer programming language than Perl.
- Perl is integrated into the source code of a web application.
- Python is a powerful programming language because of its reputation as a dynamic programming language with various applications ranging from web development to machine learning.
- Python has more support in the open-source developer community.

Perl	Python
It is a high-level, interpreter-based, dynamic programming language.	It is a high-level, general-purpose language of programming with an interpreter.
Perl can download for Unix/Linux, macOS, or Windows from https://www.perl.org/get.html .	Python can download for Unix/Linux, macOS, Windows, and other operating systems from https://www.python.org/downloads/ .
Perl's goal was to simplify the report creation process, which later underwent many changes and revisions to include many new features and capabilities.	Python aimed to make writing simple and logical code for small and large projects and applications more accessible.
When compared to Python code, Perl code is not as simple.	Python code is more straightforward and more understandable.
Perl has outstanding library support and can handle operations at the OS level using built-in functions.	To handle such operations, Python requires the assistance of third-party libraries.
The OOP support available is limited.	Python provides excellent support for object-oriented programming.
Braces are used to mark and identify code blocks.	Indentation is used to mark and identify code blocks.
Whitespaces are not crucial in Perl.	Whitespaces are significant in Python and can cause syntax errors.

(Continued)

Perl	Python
It facilitates text processing because Perl includes support for Regular expressions.	To handle Regular expressions, Python requires the use of external functions.
To end a code line in Perl, use a semicolon(;).	At the end of each code line, semicolons (;) are not required.
The file extension for Perl is .pl.	Python files have the extension. py.

PERL VS JAVA

Larry Wall invented Perl in 1987. Perl supports both object-oriented and procedural programming. It is similar to C and C++. Perl was initially designed to process text.

Java is a well-known programming language. Java is both a programming language and a computer platform. Sun Microsystems published Java in 1995, on the initiative of James Gosling. Oracle claims that java is used on 3 billion devices worldwide. It is designed to allow developers to WRITE ONCE, RUN ANYWHERE, which means that a Java application may be produced on one platform and executed on any other platform that supports the JVM.

The following are some critical distinctions between Perl and Java:

Feature	Perl	Java
Introduction	“Perl is a general-purpose, high-level programming language popular for CGI scripts.” CPanel and Bugzilla are two popular Perl projects. Initially, it was intended to replace sophisticated shell scripts.	Java is both a programming language and a platform for computing. There are still programs and websites that will not operate unless you have Java installed. It is quick, safe, and dependable.
Compiled format	Perl 6 is compiled to Parrot Bytecode, whereas Perl 5 and earlier versions are interpreted languages. It is saved with the extension. pbc.	Java applications are translated into bytecode.
Associative arrays	Perl defines associative arrays extremely succinctly.	Java bytecode may be sent across the network and then run on any system that has a JVM. It is saved with the extension.class.

(Continued)

Feature	Perl	Java
Focus	Perl excels at supporting common activities like file scanning and report creation.	There is no concise method to create associative arrays in Java. It does, however, hash implementations.
File extension	Programs in Perl are saved with .pl extension. Eg: MyFile1.pl	Programs in Java are saved with. java extension. Eg: MyFile1.java
Typed method	Perl is dynamically typed, which means that most type checking occurs during execution.	Java is statically typed, which means that most of its type checking occurs at build time.
Comments and documentation	Inline comments in the Perl are written using # E.g. #Inline-Comment in the Perl Documentation in Perl is done using = and =cut. Eg: =Perl documentation follows following syntax =cut	Single-line comments in the Java are declared using // Eg: //Single-line Comment. The Multiline comments are written using /*.....*/ Eg: /* it is a multiline comment */ Documentation in Java is done using. /**.....*/ Eg: /**Documentation in the Java*/
End of statement	Every statement in Perl must finish with a semi-colon (;)	Every statement in Java must conclude with a semi-colon (;)

PERL VS C/C++

Perl is a dynamic, high-level, interpreted, general-purpose programming language.

It was created in 1987 by Larry Wall. There is no official acronym for Perl, although “Practical Extraction and Reporting Language” is the most common.

Some programmers refer to Perl as “Pathologically Eclectic Rubbish Lister” and “Practically Everything Really Likable.” The abbreviation “Practical Extraction and Reporting Language” is often used since Perl was initially designed for text processing, such as extracting the necessary information from a specific text file and converting the text file to a new format. Both procedural and object-oriented programming is supported. C++ is a general-purpose programming language commonly utilized for competitive programming these days.

It supports imperative, object-oriented, and generic programming. C++ operates on various platforms, including Windows, Linux, Unix, and Mac.

Below are a few significant differences between Perl and C/C++:

Feature	Perl	C/C++
Driver function(main())	Perl does not require an explicit driver function.	C/C++ code must execute the main() function in order to compile.
Compilation process	Perl is a programming language that is interpreted.	C++ is a language for general-purpose object-oriented programming (OOP).
Closures	Closures containing inaccessible private data can be used as objects in Perl.	C/C++ does not allow closures, which may be considered of functions that can be stored as variables.
File extension	The .pl extension is used to store Perl scripts. For instance, perlDocument.pl	The file extensions. c and.cpp are used to save C and C++ codes, respectively. MyFile.c and myFile.cpp are two examples.
Braces	Braces must be used around the “then” part of an if statement in Perl. Ex: if (condition) { statement }	Braces are not required after if and loops in C/C++. Ex: if (condition) statement;
String declaration	Strings are declared in Perl using single quotes. Double quotes force an evaluation of the string's contents. Example: \$x = “peeksforspeeks”	To define a string in C/C++, use double quotes. Example: string s = “peeksforspeeks”
Comments	In Perl, we utilize # for inline comments. e.g. #Inline-Comment in the Perl	C/C++ uses // for the Inline comments. e.g. //Inline-Comment in the C/C++.

Perl program for adding two numbers:

```
#!/usr/bin/perl

# Perl program for adding two numbers
$choice = 13;
$choice2 = 15;
$res = add($choice, $choice2);
print "Result is $res";
```

```
# Subroutine to perform
# the addition operation
sub add
{
    ($c, $d) = @_;
    $res = $c + $d;
    return $res;
}
```

Is It Worth Learning?

Perl Excels at text manipulation. It's a fantastic language for processing logs, data munging, and pretty much anything else we can do from the command line. Despite its history as the driving force behind the monstrosity that is Perl CGI, there are new frameworks for modern web apps such as Dancer.

Perl is still a viable option for the modern programming. CPAN is still operational, and most useful modules are still maintained. Books like *Modern Perl* demonstrate how to keep Perl modern while avoiding past pitfalls.

Keeping Perl Relevant

Perl grew from a “Swiss Army Chainsaw” language to a robust scripting language for broad purposes. It has developed to the extent where it excels at certain challenges, but others should be avoided unless we enjoy the language.

We won't use it in a GUI, but we won't ignore it while dealing with data or a command line unless there's a compelling reason to do so. The language continues to be revised, and the current baseline standard is 5.8. MacOS deploys with Perl, it works on the Linux subsystem for Windows or through Strawberry Perl for scripting (though it's probably not anyone's first choice), and it's present on practically every Linux distribution (and needed by many) and every MacOS installation. It is accessible and standard on any current POSIX platform, and it can even function properly on Windows.

We used it as the foundation for a scripting language to solve the lack of a MacOS RMM tool. On an earlier Mac, there was no installation required, and it took less than a day to implement. Almost every Mac that is onboarded uses this scripting engine, which has not been updated in any significant manner since its first deployment.

What Does Perl Serve?

Perl is used for database management, system administration, and the development of GUIs. Perl was previously referred to as the “duct tape of the internet” due to its extensive functionality. It is one of the “P” choices in the web development LAMP technology stack (the other options are Python and PHP). The remaining letters represent Linux, Apache, and MySQL.

Here are the primary uses of Perl today:

- The management of systems: Perl may be used to automate or execute system management duties. These responsibilities include renaming several files in a directory or modifying a specific text component in each source file in a directory tree.
- Web development: Perl may be used to develop web apps. We may utilize frameworks such as Dancer in the Perl web development process. Web pages may also be served using Perl.
- Networking programming: We can utilize Perl’s built-in functions to create client/server applications. Perl also contains modules that make writing typical networking tasks, such as pinging distant computers, easier.
- Data management on the cloud: Many businesses have relocated their data to a public cloud nowadays. Perl can assist in cloud data management and virtual machine administration.

Perl Learning

If we want to work in technology, we should learn Perl. Companies that use Perl to build their system administration processes, network programs, and websites require Perl developers. Amazon, Roblox, Venmo, and MIT are among these companies. Perl experts hold titles like Solutions Engineer, Software Engineer, Web Developer, and Database Engineer. Perl is a great way to round out our understanding of current technologies if we want to work as a coder or system administrator in the tech industry.

Why Should We Study Perl?

Some of the reasons why we should learn Perl are as follows:

Perl is an excellent general-purpose programming language. Perl is used in a wide range of applications. These include database management, web development, system administration, and network programming.

Perl is used in intriguing ways. Bioinformatics programmers construct software for comprehending biological data sets. The expanding subject of genomics utilizes bioinformatics industry expertise. We may acquire genomic sequences and investigate population genetics using Perl.

Perl is based on linguistic concepts, and it remains an excellent text-manipulation language today. Regex, HTML parsing, and JSON manipulation come to mind. Perl can also be used to implement language processing features such as voice recognition and text-to-speech translation. Using the widespread Test Anything Protocol, you can also test applications automatically using Perl (TAP).

In conclusion, knowing Perl will provide us with intriguing options in our web development profession. Perl coders are in-demand. Once we have mastered Perl, we may utilize it to fuel our own projects or find employment. According to the TIOBE Index, a website that evaluates the popularity of programming languages based on search engine results, Perl is now ranked 17th out of 50 programming languages.

The index incorporates the results of 25 search engines. It counts a number of hits for a particular programming language, and the number of hits is used to establish that language's ranking in the index. This, combined with other data such as the frequency of job posts that mention Perl, shows that while Perl is not most popular programming language at the moment, it is still in use and hence a desirable talent to acquire.

There are over 15,000 job postings on LinkedIn and over 10,000 on Glassdoor that include Perl. Even while Perl may not be the most popular programming language today, firms continue to use it to run their web businesses, and they continue to seek individuals with this skill set.

How Long Does It Take to Learn Perl?

Give yourself three months to learn the fundamentals and begin writing programs. This assumes that we spend an hour a day learning this language.

This estimate is based on the fact that it takes about eight weeks to learn the fundamentals of Python, another scripting language that debuted around the same time as Perl (1991) and is used for many similar tasks.

However, one significant difference is that, whereas Python was designed to have a limited number of ways to perform a task, Perl was designed with the philosophy of "There's more than one way to do it." This means we may need to devote more time to learning the fundamentals of Perl.

Is Perl Difficult to Learn?

Perl can be challenging for beginners. Having prior experience with other scripting languages, on the other hand, shortens the learning curve. Learning Perl will be simple if we have some Python experience.

We must install software and interact with the command line to get started with Perl. If we've never done it before, do it before learning the Perl syntax. Perl Learn has straightforward installation instructions for all platforms.

Breakpoints of a Debugger in Perl

Controlling program execution in Perl is accomplished by instructing the debugger to execute up to a specific point in the program known as a breakpoint. These breakpoints allow the user to segment the program and search for errors.

The following debugger commands are used to create such breakpoints, as well as commands that are executed until a breakpoint is reached.

b-Command

The b-command is used to set a breakpoint in a program. This command instructs the debugger to stop the program whenever a specific line is executed.

For instance, the following command instructs the debugger to halt when it is about to execute line 12:

```
DB<13> b 12
```

(If the line is unbreakable, the debugger will return that Line 12 is not breakable.)

A program can have an unlimited number of breakpoints. When the debugger is about to execute a statement that contains a breakpoint, the program will halt.

b-command also accepts subroutine names:

```
DB<15> b valuedir
```

This sets a breakpoint at the subroutine `valuedir` very first executable statement.

The b-command can also stop a program only when a certain condition is met.

The following command, for instance, instructs the debugger to halt when it is about to execute line 12 and variable `$vardir` is equal to null:

```
DB<15> b 12 ($vardir eq "")
```

The `b` statement can be used to specify any legal Perl conditional expression.

A breakpoint can be set at any of the lines in a multiline statement.

As an example:

```
16: print ("Peeks",
17: " for Peeks here");
```

Line 16 can be used as a breakpoint, but not line 17.

c-Command

The `c`-command instructs the debugger to continue debugging until it reaches a breakpoint or the end of the program.

```
DB<15> c
```

```
main::(debugtest:12):          $vardir =~ d/^d+|\d+$/h;
```

```
DB<16>
```

When the debugger is about to reach line 12, where our breakpoint is set, the program is halted and the line is displayed, as the debugger always displays the line that is about to be executed.

The debugger generates a prompt for another debugging command here. This prompt allows us to continue the execution process one statement at a time by typing `n` or `s`, to continue the execution process by typing `c`, to set more breakpoints by typing `b`, or to perform any other debugging operation.

A temporary (one-time-only) breakpoint can be set using the `c`-command and a line number:

```
DB<15> c 12
```

```
main::(debugtest:12):          &readsubdirs($vardir);
```

The `c`-command argument 12 instructs the debugger to set a temporary breakpoint at line 12 and then continue execution. When the debugger reaches line 12, it stops the execution process, displays the line, and removes the breakpoint.

The `c`-command is used to define a temporary breakpoint when a person wants to skip a few lines of code without wasting execution time by going through the program one statement at a time. Using `c` also eliminates the need to use `b` to define a breakpoint and then delete it with the `d`-command.

L Command and Breakpoints

```
DB<18> L
```

```
4:      $count = 0;

5:      $vardir = "";

6:      while (1) {

8:          if ($vardir eq "") {

11:              $vardir =~ d/^\d+|\d+$/h;

          break if (1)
```

Lines 4–8 have been executed, and a breakpoint has been set for line 11. (Line 7 isn't included because it's a comment.) Breakpoints can distinguish from executed lines by examining the conditional expressions immediately following the breakpoint. The conditional expression, in this case, is set to (1), indicating that the breakpoint is always effective.

d and D Commands

When a breakpoint's job is complete, it can delete with the `d`-command.

```
DB<16> d 12
```

The preceding command instructs the debugger to remove the breakpoint that was set at line 12.

If no breakpoint is specified to be deleted, the debugger assumes one is defined for next line to execute and deletes it on its own.

```
main::(debugtest:12):          &readsubdirs($vardir);

DB<17> d
```

Because line 12 is the next line to be executed, the debugger automatically removes the breakpoint at line 12.

The D-command is used to remove all of the program's breakpoints.

```
DB<18> D
```

The above command removes all breakpoints set with the b command.

Exiting from a Script

Exit() evaluates the passed expression and exits the Perl interpreter, returning the value as the exit value. The exit() function does not always immediately exit but rather calls the program's end routines before terminating it. If no expression is passed to exit function, the default value is 0. The exit() function has limited uses and should not be used to exit from a subroutine. To exit a subroutine, use die or return.

Syntax:

```
exit(value)
```

Parameter: value which is to return on function call

Returns: value passed to it or 0 if the function is called without an argument

Example:

```
# Getting user's bid for
# an online auction
print "Enter bid";
$bid = <STDIN>;
```

```
# Exit function return $bid
# if bid is less than 1000
if ($bid < 1000)
{
    exit $bid;
}

else
{
    # Prints message if the bid is
    # greater than or equal to 1000
    print "\nThankyou for Participating";
}
```

Here's how the code above works:

- Step 1: Ask the user for a bid value.
- Step 2: Exit returns the bid value and terminates the program if the bid is less than 1000.
- Step 3: This message is printed if the bid is greater than or equal to 1000.

Exit function parameter passing: A parameter can pass to the exit function and be stored in a variable in the system.

It should note that the value passed to the exit function can be any random value; it does not have to be a specific value.

The following example shows how to pass a value to the exit function:

```
# Opening file
if(!open(fh,"<","Filename.txt"))
{
    # This block passing error code 56
    # to exit function indicating a file
    # could not be opened.
    print "Couldn't open file";
    exit 56;
}

# Passing success code 1 to
# exit function
exit 1;
```

The following steps show how the above program works:

- Step 1: Open a file in read-only mode.
- Step 2: If the block executes and it is unable to open a file, it calls an exit function and sends an error code of 56 to the system.
- Step 3: If the file is successfully opened, it returns 1.
- On Linux/Unix, \$? displays the values returned by an exit function. The terminal command `echo $?` displays the value returned by an exit function.

Creating Excel Files

Excel files are the commonly used office application for computer communication. It makes rows and columns of text, numbers, and calculation formulas. It's a convenient way to send reports. This demonstration is compatible with Linux, Windows, and other platforms. In Excel, rows are numbered from 1 to n... and the columns are denoted by letters ranging from A to C and so on. A1 refers to the top left corner. Padre IDE can be used to create Excel files with Perl, but we will also use Excel::Module Writer::XLSX

Perl employs the `write()` function to add content to the Excel file.

Syntax:

```
write(cell-address, content)
```

Parameters:

`cell_address`: Address of cell where content is to add.

`content`: which is to be added to worksheet.

Making an Excel File

Excel files can be created using the Perl command line, but first, the Excel::Writer::XLSX module must load.

```
#!/usr/bin/perl
use Excel::Writer::XLSX;

my $Excelbook = Excel::Writer::XLSX->new( 'PFP_Sample.
xlsx' );
my $Excelsheet = $Excelbook->add_worksheet();
```

```

$Excelsheet->write( "A1", "Hello" );
$Excelsheet->write( "A2", "PeeksForPeeks" );
$Excelsheet->write( "B1", "Next_Column" );

$Excelbook->close;

```

The program operates as follows:

Step 1: Load the Excel::Writer::XLSX module.

Step 2: Make an object called \$Excelbook that represents the entire Excel file.

Step 3: Use the write() method to insert data into the worksheet.

Step 4: Save the file as a .pl extension.

Step 5: Run our .pl file from the command line to generate an Excel spreadsheet.

Use of the Basic Formulas

Excel allows the use of various mathematical formulae to ease calculations on spreadsheets, such as balance sheets and business records.

Here is a description of two basic Excel formulas:

- **Addition:** Excel has a method called “SUM” that allows us to add values to specific cells.

Syntax:

```
=SUM(Start, End)
```

Parameter:

Start: Address of starting cell

End: Address of Ending cell

Returns: summation of values between Starting and Ending cell.

Example:

```

#!/usr/bin/perl
use Excel::Writer::XLSX;

```

```

my $Excelbook = Excel::Writer::XLSX->new( 'PFP_
Sample.xlsx' );
my $Excelsheet = $Excelbook->add_worksheet();

# Writing values at A1 and A2
$Excelsheet->write( "A1", 66 );
$Excelsheet->write( "A2", 57 );

# Adding without use of the SUM method
$Excelsheet->write( "A3", "= A1 + A2" );

# Addition of Range of cells
$Excelsheet->write( "A4", " =SUM(A1:A3)" );

```

- **Count:** This Excel function counts all the cells in a given range that contain only numeric values.

Syntax:

```
=COUNT(Start, End)
```

Returns: count of all the cells containing the numeric value

Example:

```

#!/usr/bin/perl
use Excel::Writer::XLSX;

my $Excelbook = Excel::Writer::XLSX->new( 'PFP_
Sample.xlsx' );
my $Excelsheet = $Excelbook->add_worksheet();

# Writing values
$Excelsheet->write( "A1", 6 );
$Excelsheet->write( "A2", 50 );
$Excelsheet->write( "A3", "Hello" );
$Excelsheet->write( "A4", 20 );

# Addition of Range of cells
$Excelsheet->write( "A5", "Count =" );
$Excelsheet->write( "B5", "=COUNT(A1:A4)" );

```

Adding Colors to the ExcelSheet

Colors can use in Excel Sheets to differentiate between different values. The `add_format()` method is used to specify these colors.

Syntax:

```
add_format(color=> 'colorname')
```

Example:

```
#!/usr/bin/perl
use Excel::Writer::XLSX;

my $Excelbook = Excel::Writer::XLSX->new( 'PFP_Sample.xlsx' );
my $Excelsheet = $Excelbook->add_worksheet();

# Setting value of color
my $color1 = $Excelbook->add_format(color=>
'black',);
my $color2 = $Excelbook->add_format(color=>
'yellow',);
my $color3 = $Excelbook->add_format(color=>
'blue',);

$Excelsheet->write( "A2", "Peeks", $color1 );
$Excelsheet->write( "B2", "For", $color2 );
$Excelsheet->write( "C2", "Peeks", $color3 );
$Excelbook->close;
```

Values Are Added at Specific Coordinates

Values can add at specific coordinates by specifying the cell's address where the value is to be added.

Syntax:

```
write(R,C, "value")
Parameters:
R and C are the coordinates of Row and Column,
respectively.
```

Example:

```
#!/usr/bin/perl
use Excel::Writer::XLSX;

my $Excelbook = Excel::Writer::XLSX->new( 'PFP_
Sample.xlsx' );
my $Excelsheet = $Excelbook->add_worksheet();

$Excelsheet->write( 0, 0, "Hello" );
$Excelsheet->write( 1, 0, "PeeksForPeeks" );
$Excelsheet->write( 3, 2, "Welcome!" );

$Excelbook->close;
```

Reading Excel Files

Excel sheets are one of the most popular methods for keeping office records, mainly when working on applications where non-developers and even managers can provide input to the systems in batches.

However, the problem is reading the content of a Microsoft Excel file using Perl.

CPAN provides a few modules for reading from Excel files. The spreadsheet is available: That will be capable of handling all types of spreadsheets. Other low-level libraries that read files from different versions of Excel include:

- Spreadsheet::ParseExcel Excel 95-2003 files,
- Spreadsheet::ParseXLSX Excel 2007 Open XML XLSX

Making an Excel Spreadsheet

Excel files can be created in Perl using the inbuilt module Excel::Writer::XLSX, which is used to create Excel files.

Further, the write() function is used to add content to the Excel file.

Example:

```
#!/usr/bin/perl
use Excel::Writer::XLSX;
my $Excel_book1 = Excel::Writer::XLSX->new
( 'newexcel.xlsx' );
my $Excel_sheet1 = $Excel_book1->add_worksheet();
```

```

my @data_row = (1, 2, 3, 4);
my @table_data = (
    ["l", "m"],
    ["n", "o"],
    ["p", "q"],
);
my @data_column = (1, 2, 3, 4, 5, 6, 7);

# Using write() to write values in the sheet
$Excel_sheet1->write( "A1", "Peeks For Peeks" );
$Excel_sheet1->write( "A2", "Perl|Reading Files in
the Excel" );
$Excel_sheet1->write( "A3", \@data_row );
$Excel_sheet1->write( 4, 0, \@table_data );
$Excel_sheet1->write( 0, 4, [ \@data_column ] );
$Excel_book1->close;

```

Reading from an Excel File

Spreadsheet:: is used in Perl to read an Excel file.

In a Perl script, read the module. This module exports several functions we can import or use in your Perl code script. To read from an Excel file, use the ReadData() function.

The ReadData() function takes a filename, in this case, an Excel file, but it also accepts a variety of other file types. It will load the appropriate back-end module based on the file extension before parsing the file. It generates an array reference that represents the entire file:

Example:

```

use 5.016;
use Spreadsheet::Read qw(ReadData);
my $book_data = ReadData ( 'newexcel.xlsx' );
say 'A2: ' . $book_data->[1]{A2};

```

The first element of array returned by the above code contains general information about the file. The remaining elements represent the file's other sheets. In other words, \$book_data->[1] refers to the first sheet in the "newexcel.xlsx" file. Because it is a hash reference, it can use to access the contents of the cells. \$book_data->[1]{A2} returns the hash reference for the A2 element.

Obtaining Rows from an Excel File

The arguments of the function of the Spreadsheet:::

A sheet and the number of rows to be fetched are read. The value of the rows passed in the argument is returned as an array.

The following program shows how to read the first row of the first sheet and then displays the content of each field in the row.

```
my @rowsingle = Spreadsheet::Read::row($book_data->[1], 1);
for my $x (0..$#rowsingle)
{
    say 'A'. ($x + 1). ' '.
        ($rowsingle[$x] // '');
}
```

Obtaining File Content

Obtaining a single row is insufficient. For efficient programming, we must retrieve all of the rows. The rows() function is used to accomplish this. As an argument, a sheet is passed to this function. As a matrix, it returns an array of elements or an array of references (2-D array). Each row in the spreadsheet is represented by an element in the matrix.

The following script will retrieve all rows:

```
my @rowsmulti = Spreadsheet::Read::rows($book_data->[1]);
foreach my $x (1..scalar @rowsmulti)
{
    foreach my $y (1..scalar @{$rowsmulti[$x - 1]})
    {
        say chr(64 + $x). " $x ".
            ($rowsmulti[$x - 1][$y - 1] // '');
    }
}
```

Putting Everything Together

The following Perl script demonstrates the use of all of the previously described Features of Reading an Excel file in Perl:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;
```

```

use Spreadsheet::Read qw(ReadData);

my $bookdata = ReadData('simplecreate.xlsx');

say 'A1: ' . $bookdata->[1]{A1};

# Fetching a single row
my @rowsingle = Spreadsheet::Read::row($bookdata->[1],
1);
for my $x (0..$#row)
{
    say 'A'. ($x + 1). ' ' .
        ($rowsingle[$x] // '');
}

# Fetching all the file content
my @rowsmulti = Spreadsheet::Read::rows($bookdata->[1]);
foreach my $x (1..scalar @rowsmulti)
{
    foreach my $y (1..scalar @{$rows[$x-1]})
    {
        say chr(64 + $x). " $y ".
            ($rows[$x - 1][$y - 1] // '');
    }
}

```

Perl Number Guessing Game

The number-guessing game requires us to guess the number chosen randomly by the computer within a certain number of chances.

Necessary to use the following functions:

- `rand n`: This function generates a random number from 0 to `n`. Furthermore, this function always returns a floating-point number. As a result, the result is explicitly converted to an integer value.
- `Chomp()`: `Chomp()` is a function that removes the newline character from user input.

While loop executes in the program until the number estimated by the user matches the produced number or the number of tries is fewer than the maximum number of opportunities. If the number of tries exceeds

the number of opportunities, the game ends, and the user loses. They will win if the user correctly guesses the number within the allotted number of attempts. After each guess, the computer notifies the user whether their estimate was larger or less than the actual produced number. Initially, the rand function in this code assigns x a random integer.

The (rand k) function generates a random integer between 0 and k. Since this random number is a floating-point integer, “int” is used to convert it to a whole number. x stores the integer value. The user is given a limited number of opportunities to estimate the number. If the number of opportunities exceeds, the number guessed, the user loses.

The procedure is as follows:

```
# The Number Guessing Game implementation
# using Perl

print "The Number guessing game\n";

# rand function to generate
# the random number b/w 0 to 10
# which is converted to the integer
# and store to a variable "x"
$x = int rand 10;

# variable to count correct
# number of the chances
$correct = 0;

# number of chances to be given
# to user to guess number
# number or it is the of
# inputs given by the user into
# input box here number of
# chances are 4
$chances = 4;
$n = 0;

print "Guess number (between 0 and 10): \n";

# while loop containing variable n
# which is used as a counter value
# variable chance
while($n < $chances)
```

```

{

# Enter number between 0 to 10
# Extract number from input
# and remove newline character
chomp($userinput = <STDIN>);

# To check whether the user provide
# any input or not
if($userinput != "blank")
{

    # Compare user entered number
    # with number to be guessed
    if($x == $userinput)
    {

        # if number entered by user
        # is same as generated
        # number by rand function then
        # break from the loop using loop
        # control statement "last"
        $correct = 1;
        last;
    }

    # Check if user entered
    # number is smaller than
    # the generated number
    elsif($x > $userinput)
    {
        print "Our guess was too low,";
        print " guess a higher number than
${userinput}\n";
    }

    # User entered number is
    # greater than the generated
    # number
    else

```

```

        {
            print "Our guess was too high,";
            print " guess a lower number than
${userinput}\n";
        }

        # Number of the chances given
        # to user increases by one
        $n++;

    }

    else
    {
        $chances--;
    }
}

# Check whether the user
# guessed correct number
if($correct == 1)
{
    print "We Guessed Correct!";
    print " Number was $x";
}
else
{
    print "It was actually ${x}.";
}

```

Note: In the above program, the user can change the value of the rand function to increase the range of numbers in this game and the number of chances by changing the value of the chance variable.

DATABASE MANAGEMENT USING DBI

One of the most prevalent applications of Perl is creating database apps. We can construct complex web apps with a database to handle all the data using Perl. It offers great interface support and a wide variety of database formats. Perl includes a module named DBI for connecting to and accessing a database. DBI is a database interface connecting with structured query language (SQL)-based database servers.

Generally, Perl requires two steps to access a database. The DBI module offers a database access API. DBI functions are used by software to modify

a database. A database driver (DBD) module serves as the second step in Perl database access. Each unique database management system needs its own driver. This method enables a Perl database application software to be largely independent of the database to which it will connect.

Installation: Open the terminal and enter the following command to install the DBI module:

```
perl -MCPAN -e 'install Bundle::DBI'
```

This will automatically download and install the DBI module driver, allowing Perl to connect to databases.

Database Independent Interface

DBI, as the name implies, provides an independent interface for Perl programs. This means that the Perl code is not dependent on the database that is running in the backend. The DBI module provides abstraction, which allows us to write code without worrying about the database that runs in the backend

To import functions of the Database-Independent Interface module, use the “use” pragma to import or include the module. The use of DBI pragma enables us to manipulate the database to which we are connected using the DBI module.

Syntax:

```
use DBI;
```

Accessing the Database

To connect to the specified database, use the connect() method. It requires three arguments:

- A three-value string separated by a “:” It is “DBI:mysql:test” in this case. The first value indicates that we are employing DBI. The second value specifies the database engine, which is MySQL in this case. The third value is the database name to which we want to connect.
- The username is the next argument to the connect() method. The user in this case is “root.”
- The last argument is our local system’s password. It is “password” in this scenario.

Syntax:

```
my $dbh = DBI->connect ("DBI:mysql:test",
    "root", "password") or die "Cannot connect: ".
    DBI->errstr();
```

The “or die” statement terminates the program with an error message if it is unable to connect to the database. The `errstr()` method returns a string containing any errors encountered during the database connection.

Queries to Prepare

The SQL query to execute is passed as a single parameter to the `prepare()` method. The SQL query is represented as a string that contains the SQL statement. This SQL statement is identical to the SQL statements we would use in MySQL. It returns a statement handle, which can use to execute queries.

Syntax:

```
my $sth = $dbh->prepare( " CREATE TABLE emp( id
    INT PRIMARY KEY, name VARCHAR(20), salary INT, " );
```

The query is now ready for execution. In the preceding query, we are creating a table with columns for id, name, and salary.

Executing Queries

The query written in the `prepare()` method is executed by the `execute()` method. It is not open to debate. The statement handle object created when the “prepare” statement is executed is used to call it.

Syntax:

```
$sth->execute();
```

Fetching Values from the Result

The `fetchrow()` method retrieves the next row of data from the query result. When a select query is run, the `fetchrow()` method retrieves the next row from the result. It extracts one row from the result and assigns it to variables. Using the `fetchrow()` method in a while loop, we can fetch and display all the rows in the database.

Syntax:

```
( $id, $name, $salary ) = $sth->fetchrow();
```

The three variables hold the values of each column.

The `fetchrow_array()` function returns an array containing the result row.

Syntax:

```
my @row = $sth->fetchrow_array( )
```

Disconnecting

We must disconnect the connection once all of the queries have been completed. This is accomplished through the use of `disconnect()` function. This allows the Perl script to close the connection properly. There will be no errors if we do not disconnect from the database. This is generally a good practice.

Syntax:

```
$dbh->disconnect();
```

Creating a database in MySQL

MySQL must install on our system, and we must have a basic understanding of MySQL.

- Connect to our MySQL server.
- Create a database named “test.” We’ll connect to this database, so make sure the name is “test.”
- Make sure there are no tables in this database because we will be creating a table called “emp” and inserting values into it.

Putting everything together:

We may access Perl’s database after building it in MySQL. First, we build a test table in the database with the schema: (id INTEGER PRIMARY KEY, name VARCHAR(10), salary INT, dept INT). We insert values into the table once it has been created without errors.

Once the data have been inserted, we can use the `fetchrow()` method to query the table and choose all rows to display to the user.

Example:

```
#!/usr/bin/perl -w
use DBI;

# definition of the variables

# name of database. In this case,
# the name of database in my local
# system is test.

# user in this case is root
$user = "root";
# this is the password for root
$password = "password";

# connect to MySQL database
my $dbh = DBI->connect ("DBI:mysql:test",
                        $user,
                        $password)
                        or die "Can't connect to
database: $DBI::errstr\n";

print "connected to database\n";

# test database contains a table called emp
# schema : (id INTEGER PRIMARY KEY,
#           name VARCHAR(10), salary INT, dept INT)
# let us first insert some values

# prepare query to
# create emp table
my $sth = $dbh->prepare("CREATE TABLE emp(id INT
PRIMARY KEY,
                                name
VARCHAR(10),
                                salary
INT, dept INT)");

# execute query
# now, the table is created
$sth->execute();
```

```

# prepare the query
my $sth = $dbh->prepare("INSERT INTO emp
                        VALUES(?,? ,? ,? )");

# define the variables to be inserted
# into the table
my $id = 1;
my $name = "adith";
my $salary = 1000;
my $dept = 2;

# insert these values into emp table.
$sth->execute($id, $name, $salary, $dept);

# insert some more rows into table.
$sth->execute($id + 1, $name,
            $salary + 100, $dept - 1);

# insert more rows
$sth->execute($id + 2, "Tyrion",
            $salary + 1000, $dept + 1);

print "Successfully inserted values into table\n";

# now, select all the rows from table.
my $sth = $dbh->prepare("SELECT * FROM emp");

# execute the query
$sth->execute();

# Retrieve results of a row of data and print
print "\tQuery results:\n=====
=====\\n";

# fetch the contents of table
# row by row using the fetchrow_array() function
while (my @row = $sth->fetchrow_array())
{
    print "@row\\n";
}

# if function cannot be execute, show a warning.

```

```
warn "Problem in the retrieving results",
$sth->errstr( ), "\n"
if $sth->err();

print "\n";

# select the particular columns.

# prepare the query
my $sth = $dbh->prepare("SELECT name, salary FROM
emp");

# execute the query
$sth->execute( );

# Retrieve results of a row of data and print
print "\tQuery results:\n=====
=====\\n";

while(($name, $sal) = $sth->fetchrow_array())
{
    print "Name: $name, salary: $sal\\n";
}
warn "Problem in the retrieving results", $sth-
>errstr( ), "\n"
if $sth->err( );

# end of the program
exit;
```

ACCESSING A DIRECTORY USING THE FILE GLOBBING

A directory is used in Perl to store values in the form of lists. A directory is comparable to a file in many ways. The directory, like a file, may be used to conduct various operations. These operations are used to modify an existing directory or create a new one. We may quickly open and analyze a directory using the built-in function glob.

Glob

It returns list of files that match the argument's expression. This method can print all or particular files with the extension.

Syntax:

```
@list = <*>; // Prints all files in the current
directory
@list = glob("*.pl"); // Prints all files in the
current directory with extension .pl
@list = glob('//PeeksforPeeks//Files//*');
// Prints all files in given path
```

Here are some examples of using File Globbing to access a directory.

- Getting to the script's current directory:

```
#!/usr/bin/perl -w

# Accessing files using the glob function
@files = glob('*'); # Returns list of all files
foreach $file (@files) # Loop to run through all the
files
{
    print $file. "\n"; # Print all the files
}
```

- Opening to a certain directory:

```
#!/usr/bin/perl -w

# Prints only the filename excluding path
use File::Basename;

# Returns list of all the files
@files = glob('C:/Users/PeeksForPeeks/Folder/*');
foreach $file (@files) # Loop to run through all the
files
{
    print basename($file), "\n"; # Print all the files
}
```

Hashbang or Shebang Line Use

Extraction and Reporting in Practice Language, often known as Perl, is an interpreter-based programming language. When running Perl scripts on Unix-like platforms like Linux and Mac OSX, Hashbangs or shebangs come in useful. A Hashbang line is the initial line of a Perl program and serves as a connection to the Perl binary. It enables direct execution of

Perl scripts without the need to pass the file as an input to Perl. In Perl, a Hashbang line looks like this:

```
#!/usr/bin/perl
```

A Hashbang line is so named because it begins with a Hash(#) and ends with a bang(!). A Hashbang line in Perl is very important in Perl programming. Let's get started with using this Hashbang line.

Example: Assume we have a Perl hello world program script that we will run using the terminal on a Linux system.

```
use strict;
use warnings;

print "Hello Everyone\n";
```

In the preceding code, the terminal first runs Perl, and then Perl is instructed to run the code script. An error occurs if the code script is run without first running Perl.

Try executing the following code:

```
$ hello1.pl
```

The shell we used attempted to parse the commands in the file here. However, it was unable to locate the command print in Linux/Unix. As a result, it is required to notify the shell that it is a Perl script. This is when the idea of Hashbang comes into effect. Hashbang notifies the terminal of the script.

However, before running this code, the shell's path must modify to add the current directory to existing directories. This may accomplish by running the following command:

```
$ PATH = $PATH:$(pwd)
```

The current working directory will add to the list of directories in the PATH environment variable.

Then, in the Perl script file hello1.pl, add the Hashbang line `#!/usr/bin/perl`. This line is always added at the beginning of code, i.e. the Hashbang line is the first line of the code script.

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello Everyone\n";
```

Due to the addition of the Hashbang line `#!/usr/bin/perl` to the script's first line, the code above functions correctly and generates no errors.

The script is performed in the current shell environment when it is executed. If the script begins with a hash and a bang (Hashbang) `#!`, the shell will execute the program whose path is specified on the Hashbang line (in this example, `/usr/bin/perl`), which is the default location for the Perl compiler-interpreter. Therefore, the Hashbang line contains the location of the Perl compiler-interpreter.

The error occurs when there is no Hashbang line in file, and we attempt to run it without explicitly executing Perl. The shell believes the script is written in Bash and attempts to run it appropriately, which results in errors.

Useful Math Functions

In Perl, solving certain expressions containing mathematical operations is occasionally necessary. These mathematical processes may be carried out using several built-in functions.

```
#!/usr/bin/perl

# Initialising some values for
# parameter of exp function
$A = 0;
$B = 1;

# Calling exp function
$E = exp $A;
$F = exp $B;

# Getting the value of "e" raised to
# power of given parameter.
print "$E\n";
print "$F\n";

# Calculating square root using the sqrt()
$square_root = sqrt(64);
```

```
# Printing result
print "Squareroot of 64 is: $square_root";
```

The following are some helpful Perl functions for mathematical operations:

Function	Description
<u>exp()</u>	Calculates the parameter “e” raised to the power of the real value.
<u>hex()</u>	Converts a specified hexadecimal number (of base 16) to its decimal equivalent (of base 10).
<u>srand()</u>	Helps the rand() function generates a constant value each time the program is executed.
<u>sqrt()</u>	Calculates the square root of a number.
<u>oct()</u>	Converts the supplied octal value to its decimal equivalent.
<u>rand()</u>	Returns a random fractional number ranging from 0 to the positive numeric value passed to it, or 1 if no value is specified.
<u>log()</u>	The natural logarithm of the value passed to it is returned. If called without a value, it returns \$_.
<u>int()</u>	The integer part of the provided number is returned. If no value is specified, it returns \$_.
<u>sin()</u>	Used to compute the sine of a VALUE or \$_. If VALUE is not provided.
<u>cos()</u>	Calculates the cosine of a VALUE or \$_. If VALUE is missing.
<u>atan2()</u>	Calculates the arctangent of Y/X in the range -PI to PI.
<u>abs()</u>	The absolute value of its argument is returned.

The Distinction Between Functions and Subroutines

When a code script develops in size, i.e. comprises hundreds of lines of code, it becomes tough to handle in Perl. Perl provides its users with the idea of functions and subroutines to circumvent this challenge. To make the program more readable, functions and subroutines split up complex/ large chunks of code into smaller, more concise parts.

They minimize the size of the application and debugging time by allowing us to reuse previously written code in the program. Perl functions and subroutines are used in programs to reuse code. We can utilize a function with various parameters throughout our program.

What Exactly Is a Function?

A function accepts several inputs, performs operations on them, and then returns a value.

It may be included within the programming language or provided by the user. When defining a function in Perl, the name and sequence of

statements are specified. When we wish to do a calculation later, we may “call” the function by name to execute the series of statements included in the function declaration.

Perl has several really useful built-in functions. “say” is an example of a built-in function. Perl even permits us to create our function to assist us in doing the desired task.

What Exactly Is a Subroutine?

Subroutines (or sub) allow us to provide a name to a part of code so that when we wish to utilize it again in the program, we can simply call its name.

Subroutines aid us in two ways while writing in Perl:

- First, they allowed us to reuse the code in the program, making it easier to detect and correct faults and making it faster to build applications.
- Second, they enable us to organize our code into parts. Each subroutine is in charge of the specific task.

When a piece of code is placed in a Perl subroutine, there are two options:

- When we know that the code will utilize for a calculation or action that will perform. For instance, when converting a string to a specific format or when converting an incoming data record to a hash, etc.
- When we want to divide our program’s logical components into parts to make it easier to grasp.

The following table compares the differences between a function and a subroutine:

Function	Subroutine
Basic format of a function is : \$myvalue = myfunction(parameter, parameter);	The basic format of a subroutine is : sub subroutine_name { # body of the subroutine }

(Continued)

Function

A Perl function is anything that is built into Perl. The primary reference documentation for Perl built-ins is referred to as `perlfunc`.

Perl provides us all with functions.

Functions are like subroutines in that they return a value.

A function typically performs certain computations and returns the results to the caller.

A function cannot change the value of the actual arguments.

A function produces predictable results with no side effects.

Subroutine

In Perl, a subroutine is a chunk of code that can receive inputs, execute specific actions on them, and may or may not return a useful value. They are, however, always user-defined rather than built-in.

Subroutines are pieces of code that we supply to Perl.

Subroutines carry out a task but do not return any information to the calling program.

A subroutine can change the value of the actual argument.

Such constraints do not limit a subroutine.

Example:**Function:**

```
#!/usr/bin/perl
# Program to reverse a string
# using the pre-defined function

# Creating a string
$string = "PeeksForPeeks";
print "The Original string: $string", "\n";
print "The Reversed string: ";

# Calling predefined function
print scalar reverse("$string"), "\n";
```

Subroutine:

```
#!/usr/bin/perl
# Program to reverse a string
# using subroutine

# Creating string
my $string = 'PeeksforPeeks';
```

```

print 'Original string: ', $string, "\n";
print 'Reversed using Subroutine: ',
    reverse_in_place($string), "\n";

# Creation of subroutine
sub reverse_in_place
{
    my ($string) = @_;
    my @array = split //, $string;
    my $n = scalar @array;

    for (0..$n / 2 - 1)
    {
        my $tmp = $array[$_];
        $array[$_] = $array[$n - $_ - 1];
        $array[$n - $_ - 1] = $tmp;
    }
    return join('', @array);
}

```

The Top Ten Programming Tasks for Which Perl Is Used

- **Cloud Data Management:** It's a cliché, but it's also true: all enterprises have either embraced or are in the process of adopting cloud. Do we need to access data in a public cloud but don't want to use cloud vendor's proprietary CLI tools or Web interface? There's a Perl module for it.

In reality, Perl can readily interact with all of the main cloud services to assist you with data management:

- Perl may use to interact with AWS S3 storage buckets.
- Third-party Perl utilities are also available for managing Azure Blob storage.
- Perl also supports Google Cloud Storage.
- **Perl is used to manage cloud VMs or virtual machines:** Perl may also use to manage virtual machines operating on public or private clouds. As an example:
 - Net::Amazon::EC2's Query API provides a Perl interface to Amazon's Elastic Compute Cloud (AWS EC2).

- Similarly, VMware's vSphere Perl SDK enables you to manage VMware virtual machines in any environment. However, because vSphere is a popular method for creating private clouds, Perl may be an essential tool for controlling them.

While we're unaware of any pre-built Perl solutions for interacting with virtual machine services on Azure or Google Cloud Platform, there's no reason it couldn't be done.

- **Perl is a computer language that is used to serve web pages:** Do you need a lightweight, readily adaptable Web server? One can be written in Perl in around 200 lines of code.

Although Perl version is unlikely to be used for the regular production workloads, a minimalist Perl-based Web server is excellent for systems with extremely low hardware resources, such as those found in an Internet of Things (IoT) deployment. Alternatively, examining the Perl code might be beneficial if we simply want to understand more about the foundations of how Web servers function.

- **Perl is used for speech recognition:** Speech recognition is a complicated but increasingly vital capability for various applications, including virtual assistants and chatbots. Perl can also assist here.

We may utilize Perl modules like `Google::Cloud::Speech`, which provides an interface to Google's Cloud Speech API, to enable our users to submit data via voice commands or just build a text transcript of an audio file.

While Google Cloud's API does the heavy lifting, Perl provides a simple interface for interacting with it, reducing the need to learn and deal with proprietary, vendor-specific APIs. Of course, we can use tens of thousands of additional Perl modules to help create our application.

- **Text-to-speech translation in Perl:** Perl, like voice recognition, may be used to convert text to speech utilizing Google's translate service to create speech from any language text.

While text-to-speech may not seem as thrilling as speech-to-text, it may help us add useful capabilities to our application, such as screen reading for accessibility, enabling multi-tasking, or just assisting in teaching a new language.

- **TAP for software testing:** If we wish to release software on a continuous and automated basis, we must likewise test it on a continuous

and automated basis. Perl has multiple best-in-class testing methods, beginning with the most well-known and frequently used Test Anything Protocol, often known as TAP::Harness.

TAP essentially offers a text-based interface between Perl testing modules. Still, it's so reliable and simple to use that it now has implementations in C, C++, Python, PHP, Perl, Java, JavaScript, and other languages.

- **Use it to perform system administration tasks:** Perl has long followed significantly in the Unix/Linux environment, where administrators frequently use it for scripting typical system management duties.

However, Perl may also be helpful in Windows-centric systems. Perl allows us to deal with Active Directory and even the Windows registry. Do we need to track the software installed on our users' computers? Or should periodic updates/maintenance be performed when the system boots? Perl may be the ideal technique to script such administrative operations on Windows and Linux.

- **BioPerl for bioinformatics:** Although bioinformatics is unlikely to be at the top of most developers' lists of Perl usage, there is a community dedicated to it called BioPerl. Perl-based bioinformatics and genomics solutions range from obtaining genomic sequences to investigating population genetics.

Perl is ideal for log management: Because of the distribution of modern computer systems and the way they combine so many different types of components, modern log management has become a science unto itself.

Perl is useful for interacting with most log management solutions, from standards like syslog to proprietary systems like Papertrail. Of course, we may develop our own Perl scripts for aggregating and parsing logs, which is handy in instances when a fully customized log management solution is required.

- **Perl is excellent for text manipulation:** Text manipulation should come as no surprise as Perl's most common application.

Perl has been the language of choice for regex, HTML parsing, and JSON manipulation for over three decades.

No other programming language provides more effective or user-friendly text manipulation features than Python.

A wealth of Perl community modules is available to aid with almost any task involving text manipulation, extraction, and transformation. Many of these have been pre-compiled in our Perl Text Processing environment, which we can download and install for free via our State Tool CLI.

Bibliography

1. Perl – Introduction. https://www.tutorialspoint.com/perl/perl_introduction.htm, accessed on June 1, 2022.
2. Perl. <https://www.techtarget.com/whatis/definition/Perl>, accessed on June 1, 2022.
3. Introduction to Perl. <https://www.geeksforgeeks.org/introduction-to-perl/>, accessed on June 1, 2022.
4. What is Perl? <https://www.educba.com/what-is-perl/>, accessed on June 1, 2022.
5. What is Perl? <http://www.cs.unc.edu/~jbs/resources/perl/perl-basics/>, accessed on June 2, 2022.
6. Perl OOP. <https://www.perltutorial.org/perl-oop/>, accessed on June 2, 2022.
7. Perl Tutorial: Variable, Array, Hashes with Programming Example. <https://www.guru99.com/perl-tutorials.html>, accessed on June 2, 2022.
8. Perl – Operators. https://www.tutorialspoint.com/perl/perl_operators.htm, accessed on June 3, 2022.
9. Perl Operator Types. <https://www.javatpoint.com/perl-operator-types>, accessed on June 3, 2022.
10. Perl Operators – Complete Guide. <https://beginnersbook.com/2017/02/perl-operators-complete-guide/>, accessed on June 3, 2022.
11. Perl | Decision Making (if, if-else, Nested-if, if-elsif ladder, unless, unless-else, unless-elsif). <https://www.geeksforgeeks.org/perl-decision-making-if-if-else-nested-if-if-elsif-ladder-unless-unless-else-unless-elsif/>, accessed on June 4, 2022.
12. Perl Conditional Statements - IF...ELSE. https://www.tutorialspoint.com/perl/perl_conditions.htm, accessed on June 4, 2022.
13. Perl If Statement. <https://www.perltutorial.org/perl-if/>, accessed on June 4, 2022.
14. Conditional Decisions. https://www.learn-perl.org/en/Conditional_Decisions, accessed on June 5, 2022.
15. What are packages and modules in Perl? <https://www.educative.io/answers/what-are-packages-and-modules-in-perl>, accessed on June 5, 2022.
16. Modules and Packages in Perl. <https://www.codesdope.com/perl-modules-and-packages/>, accessed on June 6, 2022.
17. Perl Fundamentals. <https://www.udemy.com/course/perl-fundamentals/>, accessed on June 6, 2022.
18. Perl Tutorial. <https://www.tutorialspoint.com/perl/index.htm>, accessed on June 6, 2022.

19. Perl Tutorial. <https://www.perltutorial.org/>, accessed on June 6, 2022.
20. Chapter 10. Packages. https://docstore.mik.ua/oreilly/perl4/prog/ch10_01.htm, accessed on June 6, 2022.
21. Perl – Variables. https://www.tutorialspoint.com/perl/perl_variables.htm, accessed on June 6, 2022.
22. Perl | Variables. <https://www.geeksforgeeks.org/perl-variables/>, accessed on June 7, 2022.
23. Set Perl Variables. https://www.webassign.net/manual/instructor_guide/t_i_setting_perl_variables.htm, accessed on June 7, 2022.
24. Object Oriented Programming in PERL. https://www.tutorialspoint.com/perl/perl_object_oriented.htm, accessed on June 7, 2022.
25. Perl Object Oriented – javatpoint. <https://www.javatpoint.com/perl-object-oriented>, accessed on June 7, 2022.
26. Object Oriented Programming (OOPs) in Perl. <https://www.geeksforgeeks.org/object-oriented-programming-oops-in-perl/>, accessed on June 7, 2022.
27. Inheritance in Perl. <https://www.codesdope.com/perl-inheritance/>, accessed on June 7, 2022.
28. Perl Subroutine. <https://www.perltutorial.org/perl-subroutine/>, accessed on June 7, 2022.
29. Perl Functions and Subroutines. <https://www.javatpoint.com/perl-functions-and-subroutines>, accessed on June 7, 2022.
30. What Are Subroutines in Perl? <https://www.educative.io/answers/what-are-subroutines-in-perl>, accessed on June 7, 2022.
31. Perl | Subroutines or Functions. <https://www.geeksforgeeks.org/perl-subroutines-or-functions/>, accessed on June 7, 2022.
32. Perl – Subroutines. https://www.tutorialspoint.com/perl/perl_subroutines.htm, accessed on June 7, 2022.
33. What Are References in Perl? <https://www.educative.io/answers/what-are-references-in-perl>, accessed on June 8, 2022.
34. Learning Perl Objects, References, and Modules. <https://www.oreilly.com/library/view/learning-perl-objects/0596004788/ch04.html>, accessed on June 8, 2022.
35. Perl | Mutable and Immutable Parameters. <https://www.geeksforgeeks.org/perl-mutable-and-immutable-parameters/>, accessed on June 8, 2022.
36. Chapter 9. Arrays and Lists. <https://www.oreilly.com/library/view/think-perl-6/9781491980545/ch09.html>, accessed on June 8, 2022.
37. Perl - Regular Expressions. https://www.tutorialspoint.com/perl/perl_regular_expressions.htm, accessed on June 8, 2022.
38. Perl Regular Expressions. <https://perldoc.perl.org/perlre>, accessed on June 8, 2022.
39. Perl Regular Expressions. <https://www.javatpoint.com/perl-regular-expression>, accessed on June 8, 2022.
40. Perl Tutorial. Regular Expressions (Regex), File IO and Text Processing. https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/Perl2_Regexe.html, accessed on June 8, 2022.
41. Perl Regular Expressions. <https://oval.mitre.org/language/about/perlre.html>, accessed on June 9, 2022.

42. Perl | Regular Expressions. <https://www.geeksforgeeks.org/perl-regular-expressions/>, accessed on June 9, 2022.
43. Scalar Data and Operators in Perl. <https://www.informit.com/articles/article.aspx?p=30227&seqNum=7>, accessed on June 9, 2022.
44. Perl Programming/User input-output. https://en.wikibooks.org/wiki/Perl_Programming/User_input-output, accessed on June 9, 2022.
45. Perl | File I/O Functions. <https://www.geeksforgeeks.org/perl-file-i-o-functions/>, accessed on June 9, 2022.
46. File I/O in Perl. <https://www.codesdope.com/perl-file-io/>, accessed on June 10, 2022.
47. File I/O in Perl. <https://www.codesdope.com/perl-file-io/>, accessed on June 10, 2022.
48. Perl | File I/O Functions. <https://www.geeksforgeeks.org/perl-file-i-o-functions/>, accessed on June 10, 2022.
49. Perl - Data Types. https://www.tutorialspoint.com/perl/perl_data_types.htm, accessed on June 10, 2022.
50. Perl | Data Types. <https://www.geeksforgeeks.org/perl-data-types/>, accessed on June 11, 2022.
51. Data Types in Perl. <https://beginnersbook.com/2017/02/data-types-in-perl/>, accessed on June 11, 2022.
52. Data Types and Variables. <https://www.oreilly.com/library/view/perl-in-a/1565922867/ch04s02.html>, accessed on June 11, 2022.
53. Perl Data Types. <https://www.educba.com/perl-data-types/>, accessed on June 12, 2022.
54. Perl - File I/O. https://www.tutorialspoint.com/perl/perl_files.htm, accessed on June 12, 2022.
55. Perl Open File. <https://www.perltutorial.org/perl-open-file/>, accessed on June 13, 2022.
56. File I/O in Perl. <https://www.codesdope.com/perl-file-io/>, accessed on June 13, 2022.
57. Control Structures in Perl Control Structures in Perl; http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci132/slides/Lesson_15.pdf, accessed on June 13, 2022.
58. Introducing Perl. <https://www.informit.com/articles/article.aspx?p=29028&seqNum=3>, accessed on June 13, 2022.
59. File Handling in Perl. <https://www.educba.com/file-handling-in-perl/>, accessed on June 13, 2022.
60. Perl | File Handling Introduction. <https://www.geeksforgeeks.org/perl-file-handling-introduction/>
61. Perl - Hashes. https://www.tutorialspoint.com/perl/perl_hashes.htm, accessed on June 13, 2022.
62. Perl | Hashes. <https://www.geeksforgeeks.org/perl-hashes/>, accessed on June 13, 2022.
63. Perl Hash. <https://www.perltutorial.org/perl-hash/>, accessed on June 13, 2022.

64. Perl Hashes. <https://www.javatpoint.com/perl-hashes>, accessed on June 13, 2022.
65. Hashes of Arrays (Programming Perl). https://docstore.mik.ua/orelly/perl2/prog/ch09_02.htm, accessed on June 14, 2022.
66. Perl|ListContextSensitivity.<https://www.geeksforgeeks.org/perl-list-context-sensitivity/#:~:text=In%20Perl%2C%20function%20calls%2C%20terms,gives%20the%20list%20of%20elements>, accessed on June 14, 2022.
67. Perl String. <https://www.perltutorial.org/perl-string/>, accessed on June 14, 2022.
68. Perl String. <https://www.javatpoint.com/perl-string>, accessed on June 14, 2022.
69. Perl | String Functions (length, lc, uc, index, rindex). <https://www.geeksforgeeks.org/perl-string-functions-length-lc-uc-index-rindex/>, accessed on June 14, 2022.
70. What Are Perl String Literals? <https://www.tutorialspoint.com/what-are-perl-string-literals#:~:text=Strings%20are%20sequences%20of%20characters,strings%20and%20double%2Dquoted%20strings>, accessed on June 15, 2022.
71. String Literals in Perl. https://www.perlmonks.org/?node_id=945, accessed on June 15, 2022.
72. What Are Special Literals in Perl 5.3.4? <https://www.educative.io/answers/what-are-special-literals-in-perl-534>, accessed on June 15, 2022.
73. perldata - Perl Data Types. <https://perldoc.perl.org/perldata#:~:text=function's%20calling%20context.-,Scalar%20values,form%20to%20another%20is%20transparent>, accessed on June 15, 2022.
74. Perl | Scalars. <https://www.geeksforgeeks.org/perl-scalars/>, accessed on June 15, 2022.
75. Perl – Scalars. https://www.tutorialspoint.com/perl/perl_scalars.htm, accessed on June 15, 2022.
76. Perl - Socket Programming. https://www.tutorialspoint.com/perl/perl_socket_programming.htm, accessed on June 15, 2022.
77. Perl socket - Programming Sockets in Perl. <https://zetcode.com/perl/socket/>, accessed on June 15, 2022.
78. Perl | Socket Programming. <https://www.geeksforgeeks.org/perl-socket-programming/#:~:text=Socket%20programming%20in%20Perl%20is,shows%20them%20using%20socket%20connection>, accessed on June 16, 2022.
79. Perl Reference. <https://www.perltutorial.org/perl-reference/>, accessed on June 16, 2022.
80. Perl | References. <https://www.geeksforgeeks.org/perl-references/>, accessed on June 16, 2022.
81. Perl – References. https://www.tutorialspoint.com/perl/perl_references.htm, accessed on June 16, 2022.
82. What are references in Perl?. <https://www.educative.io/answers/what-are-references-in-perl>, accessed on June 16, 2022.

83. Error Handling in Perl. <https://www.geeksforgeeks.org/error-handling-in-perl/>, accessed on June 17, 2022.
84. Perl - Error Handling. https://www.tutorialspoint.com/perl/perl_error_handling.htm, accessed on June 17, 2022.
85. Object Oriented Exception Handling in Perl. <https://www.perl.com/pub/2002/11/14/exception.html>, accessed on June 17, 2022.
86. Perl Error Handling. <https://www.javatpoint.com/perl-error-handling>, accessed on June 17, 2022.

Index

A

Add assignment, [38](#)
Addition, [30](#)
Advantages of Perl, [239–240](#)
Anchors, [169](#)
AND logical operator, [132](#)
API, *see* [Application programming interface](#)
Application of Perl, [4, 6](#)
Application programming interface (API),
[151](#)
Arithmetic operators, [30](#)
 addition, [30](#)
 division, [30](#)
 exponent operator, [31](#)
 modulus operator, [31](#)
 multiplication, [30](#)
 subtraction, [30](#)
Array variables, [18–19, 42, 43, 45, 46](#)
Assignment operators, [37–40](#)

B

Backtracking in regular expression,
[156–158](#)
Base class and derived class,
[197–198](#)
Base classes, [196](#)
b-command, [248–249](#)
BEGIN block, [60](#)
Benefits of Perl, [6](#)
Binary left shift operator, [36](#)
Binary numbers, [63–64](#)
Binary right shift operator, [36](#)
BioPerl for bioinformatics, [278](#)

Bitwise operators, [35–37](#)
Bless method, [185, 189](#)
Block, [14–15](#)
Boolean expression, [40](#)
Boolean values in Perl, [25–29](#)
Break keyword, [97](#)
Breakpoints of debugger in Perl, [248](#)
 b-command, [248–249](#)
 c-command, [249–250](#)
 d and D commands, [250–251](#)
 L command and breakpoints, [250](#)

C

Calculator module, [58–59](#)
“Call by reference,” [223](#)
“Call by value,” [223](#)
Capturing, [172](#)
C/C++, Perl vs, [243–244](#)
c-command, [249–250](#)
CGI scripts, *see* [Common Gateway Interface scripts](#)
Challenges of learning Perl, [248](#)
Character classes, [168](#)
chdir() function, [67](#)
chomp() function, [75, 260](#)
Class, [178, 180, 181](#)
 creating, [182](#)
 defining, [182](#)
Class instance, creating, [182–183](#)
Cloud data management, [276](#)
Cloud VMs/virtual machines, managing,
[276–277](#)
Comma-separated value (CSV) file
 reading, [125](#)
 character escaping a comma,
[127–128](#)

- fields with newlines embedded, [129–131](#)
- TEXT::CSV, installation of, [128–129](#)
- use of split() for data extraction, [125–127](#)
- Comments, [5–6](#), [13–14](#)
 - multi-line string as, [13–14](#)
 - single-line comments, [13](#)
- Common Gateway Interface (CGI)
 - scripts, [4](#)
- Complement operator, [36](#)
- Comprehensive Perl archive network (CPAN), [6](#), [54](#), [245](#), [257](#)
- Conditional statement, [98](#)
- connect() method, [264](#)
- Constructors, [189–191](#)
- Control flow in Perl, [77](#)
 - decision-making in Perl, [77](#)
 - if else statement, [79–81](#)
 - if elsif else ladder statement, [83–85](#)
 - if statement, [78–79](#)
 - nested if statement, [81–83](#)
 - unless else statement, [86–88](#)
 - unless elsif statement, [88–90](#)
 - unless statement, [85–86](#)
 - given-when statement, [97](#)
 - nested given-when statement, [98–100](#)
 - goto statement, [100–103](#)
 - last keyword, [106–107](#)
 - loops in Perl, [90](#)
 - do...while loop, [93–94](#)
 - foreach loop, [92](#)
 - for loop, [90–92](#)
 - infinite while loop, [93](#)
 - nested loops, [95–97](#)
 - until loop, [94–95](#)
 - while loop, [92–93](#)
 - next operator, [103–104](#)
 - redo operator, [105](#)
- Control statements, [77](#), [97](#)
- CPAN, *see* [Comprehensive Perl archive network](#)
- CRUD operations, [69](#)
- CSV file reading, *see* [Comma-separated value file reading](#)

D

- d and D commands, [250–251](#)
- Data abstraction, [180](#)
- Database driver (DBD) module, [264](#)
- Database Independent Interface (DBI), [264](#)
- Database management using DBI, [263](#)
 - accessing the database, [264–265](#)
 - disconnecting, [266](#)
 - executing queries, [265](#)
 - fetching values from the result, [265–266](#)
 - MySQL, creating a database in, [266–269](#)
 - queries to prepare, [265](#)
- Data member, [182](#)
- Data types, [18](#), [45](#)
 - arrays, [18–19](#)
 - hashes, [19–20](#)
 - scalars, [18](#)
- DBD module, *see* [Database driver module](#)
- DBI, *see* [Database Independent Interface](#)
- d character, [150](#)
- Debugger in Perl, [248](#)
 - b-command, [248–249](#)
 - c-command, [249–250](#)
 - d and D commands, [250–251](#)
 - L command and breakpoints, [250](#)
- Decision-making in Perl, [77](#)
 - if else statement, [79–81](#)
 - if elsif else ladder statement, [83–85](#)
 - if statement, [78–79](#)
 - nested if statement, [81–83](#)
 - unless else statement, [86–88](#)
 - unless elsif statement, [88–90](#)
 - unless statement, [85–86](#)
- Dereferencing, [231–232](#)
- Derived classes, [196](#)
- Destructors, [192](#)
- Die function, [122–123](#)
- Directory, [64](#), [269](#)
 - BioPerl for bioinformatics, [278](#)
 - closing, [68](#)
 - cloud data management, [276](#)
 - cloud VMs/virtual machines, managing, [276–277](#)
 - deleting, [68–69](#)

- existing directory, opening, 65
 - function, 273–274
 - glob, 269–270
 - hashbang/shebang line use, 270–272
 - math functions, 272–273
 - modifying directory path, 67–68
 - new directory, making, 65
 - reading directory in scalar and list context, 65–67
 - speech recognition, Perl for, 277
 - subroutine, 274
 - system administration tasks, 278
 - TAP for software testing, 277–278
 - text manipulation, 278–279
 - text-to-speech translation in Perl, 277
 - web pages, serving, 277
 - DIRHANDLE, 68
 - Disadvantages of Perl, 6, 240
 - Division, 30
 - Division assignment, 38
 - Do...while loop, 93–94
 - Download, Perl, 8
 - Duration of learning Perl, 247
 - Dynamic attributes, passing, 191–192
 - Dynamic method dispatch, 196
-
- E
-
- “ee” modifier in regular expression, 161–164
 - “e” modifier in regular expression, 159–161
 - Encapsulation in Perl, 180, 203–205
 - END block, 60
 - End of file (EOF), 119
 - EOF, *see* End of file
 - Error handling and error reporting, 122
 - die function, 122–123
 - warn function, 123
 - Evolution of Perl, 1–2
 - Excel files, 253, 257
 - basic formulas, use of, 254–255
 - Excel spreadsheet, making, 257–258
 - file content, obtaining, 259
 - making, 253–254
 - obtaining rows from, 259
 - putting everything together, 259–260
 - reading from, 258
 - values, 256–257
 - Excel Sheet, adding colors to, 256
 - Exclusive lock, 134–135
 - Executing Perl program, 11
 - Unix/Linux, 11
 - Windows, 11
 - exit() function, 251
 - Exponent assignment, 39
 - Exponent operator, 31
 - Expressions, 12
-
- F
-
- False values, 27–28
 - fcntl() system function, 136
 - Features of Perl, 3, 238
 - fetchrow() method, 265, 266
 - fetchrow_array() function, 266
 - Fibonacci series, 227
 - File
 - appending to, 123–125
 - exception handling in, 119
 - throwing an exception, 119
 - warning, giving, 120
 - opening, 117
 - reading, 117
 - FileHandle operator, 117–118
 - getc function, 118
 - read function, 118–119
 - reading more than one line at a time, 119
 - File, writing to, 120
 - error handling and error reporting, 122
 - die function, 122–123
 - warn function, 123
 - print() function, 120–122
 - File::Slurp module, 136
 - installation of, 136–137
 - FileHandle operator, 117–118
 - File handling, 109
 - CSV (comma-separated value) file
 - reading, 125
 - character escaping a comma, 127–128
 - fields with newlines embedded, 129–131

TEXT::CSV, installation of, [128–129](#)
 use of `split()` for data extraction,
 [125–127](#)
 redirecting output, [115–116](#)
 slurp module, [136–139](#)
 useful functions, [139–140](#)
 using `FileHandle` to read and write to a
 file, [110–111](#)
 various modes, [111–115](#)
 File locking, [133](#)
 `flock()`, [134–136](#)
 vs `lockf()`, [136](#)
 File test operators, [131–133](#)
 Fixed point, [61](#)
 Floating-point numbers, [61](#)
`flock()`, [134–136](#)
 vs `lockf()`, [136](#)
 Foreach loop, [92](#)
 For loop, [90–92](#)
 Function, [15](#), [273–274](#)
 Function signature, [214–215](#)
 argument different than, [215–216](#)
 defining subroutines, [214](#)
 difference in number of arguments,
 [216–217](#)
 subroutine/function with defined
 signature, [215](#)

G

`\G` Assertion, [166–167](#)
`getc` function, [118](#)
 “`get_mileage`” method, [196](#)
 Get-set methods, [187–189](#)
 Given-when statement, [98](#)
 Given-when statement, [97](#)
 nested given-when statement, [98–100](#)
`Glob`, [269–270](#)
 Global variables
 scope of, [48–50](#)
 in subroutines, [212–213](#)
 Gosling, James, [242](#)
`Goto` statement, [100–103](#)
 Graphical user interface (GUI)
 programming, [2](#), [237](#)
 Grouping and capturing, [172](#)
 GUI programming, *see* [Graphical user](#)
 [interface programming](#)

H

Hashbang/shebang line use,
 [270–272](#)
 Hashes, [19–20](#)
 Hash variables, [42](#), [43](#), [46–47](#)
 Hello Everyone! program, [9–10](#)
 Hexadecimal numerals, [62–63](#)
 Hierarchical Inheritance, [197](#)
 High-level language, Perl as, [2](#)
 History of Perl, [238](#)

I

IDEs, *see* [Integrated development](#)
 [environments](#)
 If else statement, [79–81](#)
 If elsif else ladder statement, [83–85](#)
 If statement, [78–79](#)
 Immutable parameters, [222–223](#)
 Implementation of Perl, [4](#)
 Infinite while loop, [93](#)
 Inheritance, [179](#), [199–200](#)
 Init statement, [91](#)
 Input and output in Perl, [71](#)
 print operator, [73–74](#)
 `print()` operator, [71–72](#)
 `say()` function, [73](#)
 STDIN, [74–75](#)
 Installation of Perl, [8](#)
 Linux installation, [9](#)
 macOS installation, [9](#)
 Windows installation, [8](#)
 Integers, [61](#)
 Integrated development environments
 (IDEs), [2](#)
 Interactive mode, [22–23](#)
 Internet of Things (IoT), [277](#)
 Interpreted language, Perl as, [239](#)
 Interpreter, [10](#)
 IoT, *see* [Internet of Things](#)

J

Java, Perl vs, [242–243](#)

K

Keywords, [17](#)

L

-
- LABEL name, 101
 - Last keyword, 106–107
 - L command and breakpoints, 250
 - Learning Perl, 245, 246
 - challenges of, 248
 - duration of, 247
 - Lexical variables, 48
 - scope, 50–52
 - Licensing for Perl, 238–239
 - Linux
 - installation, 9
 - installation and configuration of Perl on, 7
 - Local variable in subroutines, 212–213
 - lockf() vs flock(), 136
 - Logical operators, 34–35
 - Loops in Perl, 15–16, 90
 - do...while loop, 93–94
 - foreach loop, 92
 - for loop, 90–92
 - infinite while loop, 93
 - nested loops, 95–97
 - termination, 91
 - until loop, 94–95
 - while loop, 92–93

M

-
- Macintosh, 7
 - macOS
 - installation, 9, 245
 - installation and configuration of Perl on, 7
 - Math functions, 272–273
 - Metacharacters, 170
 - Method, 179
 - Method overriding in OOPs, 192–196
 - MODE function, 65
 - Modifiers, 171, 172
 - Modules in Perl, 54
 - importing and using, 55
 - making, 54–55
 - predefined modules, making use of, 56
 - utilizing module variables, 55–56
 - Modulus assignment, 39
 - Modulus operator, 31

- Multilevel Inheritance, 198
- Multi-line statements, 14
- Multi-line string as comments, 13–14
- Multiple subroutines, 225
 - “multi” keyword, use of, 226–227
 - subroutine definition, 225–226
- Multiplication, 30
- Multiply assignment, 38
- Mutable parameters, 222
- MySQL, creating a database in, 266–269

N

-
- Need for Perl, 2–3
 - Nested given-when statement, 98–100
 - Nested if statement, 81–83
 - Nested loops, 95–97
 - Next operator, 103–104
 - Non-blocking lock, 135
 - Number and its types in Perl, 60–64
 - Number guessing game, 260–263

O

-
- Object-oriented programming (OOP), 177
 - class, 181
 - creating, 182
 - defining, 182
 - class instance, creating, 182–183
 - constructors, 189–191
 - data member, 182
 - destructors, 192
 - dynamic attributes, passing, 191–192
 - encapsulation in, 203–205
 - inheritance in, 196
 - base class and derived class, 197–198
 - implementing inheritance in Perl, 199–200
 - Multilevel Inheritance, 198
 - methods in, 186
 - get-set methods, 187–189
 - types of methods in Perl, 187
 - overriding in, 192–196
 - polymorphism in, 200–202
 - Objects, 179, 181, 184–186
 - creating, 183–184
 - making use of, 182

Octal numbers, 63
 One-liner mode, 24
 OOP, *see* [Object-oriented programming](#)
 Open function, 117
 Operators, 29

- arithmetic operators, 30
 - addition, 30
 - division, 30
 - exponent operator, 31
 - modulus operator, 31
 - multiplication, 30
 - subtraction, 30
- assignment operators, 37–40
- bitwise operators, 35–37
- logical operators, 34–35
- in regular expression, 143–146
- relational operators, 31–34
- ternary operator, 40–41

 Overriding in OOPs, 192–196

P

Packages, Perl, 57

- begin and end block, 60
- making use of a Perl module, 58
- module variables, utilizing, 59–60
- Perl module declaration, 57–58
- using a different directory to access a package, 58–59

 Package variables, 52–54
 PATTERN, 144–145
 Polymorphism in OOPs, 179, 196, 200–202
 pos() function in regular expression, 164–167
 POSIX platform, 245
 Pragma, 10
 Predefined modules, making use of, 56
 Preinstalled version of Perl, 7
 Primary uses of Perl, 246
 print() function, 10, 120–122
 print() operator, 71–72
 Print function, 72, 110–111, 116
 Print operator, 73–74
 Private variables, 50–52
 Programming in Perl, 4, 7

- applications, 6

- benefits of Perl, 6
- comments, 5–6
- disadvantages of Perl, 6

 Python, Perl vs, 240–242

Q

Quantifiers in regular expression, 152, 170–171, 172

- quantifier table, 153–156

R

rand k function, 261
 rand n function, 260
 Read, Evaluate, Print, Loop (REPL), 22
 Read-Append mode, 115
 ReadData() function, 258
 Read function, 118–119
 Read-Write mode, 113–115
 Recursion, 234–236
 Redo operator, 105
 References, 229

- dereferencing, 231–232
- making, 229–230
- pass by, 232–234

 Regular expressions (Regex), 141

- backtracking in, 156–158
- cheat sheet, 167
 - anchors, 169
 - character classes, 168
 - grouping and capturing, 172
 - metacharacters, 170
 - modifiers, 171
 - quantifiers, 170–171, 172
 - white space modifiers, 171
- “ee” modifier in, 161–164
- “e” modifier in, 159
 - substitution operation, 160–161
- operators in, 143–146
- pos() function in regular expression, 164–167
- quantifiers in, 152
 - quantifier table, 153–156
- regex character classes, 147–149
- searching in a file using, 172
 - regular search, 173–174

- wild cards, use of, 175–176
- word boundary, using, 174–175
- special character classes in, 150–152
- Regular search, 173–174
- Relational operators, 31–34
- Relevancy of Perl, 245
- REPL, *see* Read, Evaluate, Print, Loop
- return() function, 228–229
- Reusability, 179–180
- rows() function, 259

S

- say() function, 73
- Scalars, 18
- Scalar variables, 42–43, 45–46
- Script, exiting from, 251–253
- Script mode, 23–24
- Shared lock, 134–135
- Simple assignment, 38
- Single-line comments, 13
- Slurp module, 136–139
- Software testing, TAP for, 277–278
- Speech recognition, Perl for, 277
- split() function, 125–127
- SQL, *see* Structured query language
- Statement execution, 91
- Statements, 14
- Static method, 187
- STDIN, 74–75
- Strawberry Perl, 245
- Structured query language (SQL), 263
- Studying Perl, 246–247
- Subclass, 179
- Subroutine arity, 226
- Subroutines, 15, 207, 214–215, 274
 - calling, 208
 - determining, 208
 - function signature in Perl, 214–217
 - global variable in, 212–213
 - immutable parameters, 222–223
 - local variable in, 212–213
 - multiple subroutines, 225
 - “multi” keyword, use of, 226–227
 - subroutine definition, 225–226
 - mutable parameters, 222
 - passing complex parameters to, 217–222

- passing hashes to, 210
- passing lists to, 210–211
- passing parameters to, 208–209
- recursion, 234–236
- references, 229
 - dereferencing, 231–232
 - making, 229–230
 - pass by, 232–234
- return() function, 228–229
- returning a value from, 211–212
- traits, 223–225
- varying number of parameters in
 - subroutine call, 213–214
- Subtract assignment, 38
- Subtraction, 30
- Sun Microsystems, 242
- Superclass, 179
- Switch-case in Perl, 97
- Syntax of Perl program, 11
 - block, 14–15
 - comments, 13–14
 - expressions, 12
 - functions/subroutines, 15
 - keywords, 17
 - loops, 15–16
 - statements, 14
 - variables, 12
 - whitespaces and indentation, 16–17
- System management, 2

T

- TAP for software testing, 277–278
- Ternary operator, 40–41
- TEXT::CSV, installation of, 128–129
- Text manipulation, 278–279
- Text-processing, 2
- Text-to-speech translation in Perl, 277
- Traits, 223–225
- True values, 25–27

U

- Unicode character classes, 152
- Unix/Linux, 11
- Unless else statement, 86–88
- Unless elsif statement, 88–90

Unless statement, [85–86](#)
 Until loop, [94–95](#)
 Uses of Perl, [246](#)

V

Variables, [12](#), [41](#), [44](#)
 array, [46](#)
 context, [47–48](#)
 creating, [45](#)
 declaration of, [42](#)
 Hash, [46–47](#)
 interpolation, [43–44](#)
 modification of, [42–43](#)
 naming of, [41–42](#)
 scalar, [45–46](#)
 scope of, [48](#)
 global variables, [48–50](#)
 lexical variables, [50–52](#)
 package variables, [52–54](#)
 Virtual method, [187](#)

W

Wall, Larry, [1–2](#), [237](#), [238](#), [242](#), [243](#)
 Warn function, [123](#)
 Web and Perl, [2](#), [239](#)
 Web pages, serving, [277](#)
 While loop, [92–93](#)
 White space modifiers, [171](#)
 Whitespaces, [152](#)
 and indentation, [16–17](#)
 Wild cards in regular expression, [175–176](#)
 Windows, [11](#)
 installation, [8](#)
 installation and configuration of Perl
 on, [7](#)
 Word boundary in regex search, [174–175](#)
 write() function, [257](#)
 Writing methods, Perl code, [21](#)
 interactive mode, [22–23](#)
 one-liner mode, [24](#)
 script mode, [23–24](#)